AD-A040 685 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES.(U)
MAR 77 T B HICKEY DAAB07-72-C-0259
UNCLASSIFIED R-761 NL

1 OF 2
AD
A040685

REPORT R-761 MARCH, 1977 UILU-ENG 77-2208

CSL COORDINATED SCIENCE LABORATORY

SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES

THOMAS BUTLER HICKEY

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES.		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report.
		6. PERFORMING ORG. REPORT NUMBER (14) R-761, UILU-ENG-77-2208
7. AUTHOR(s) (10) Thomas Butler Hickey		8. CONTRACT OR GRANT NUMBER(s) (15) DAAB 07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 143 p.
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE (11) Mar[redacted]77
		13. NUMBER OF PAGES 131
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Superimposed Coding
Sequential and Inverted Files
Data Bases

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The relative efficiency of three computer search algorithms was compared for searching large bibliographic files with Boolean search strategies. The sequential and inverted files represent the two most common file structures used today for bibliographic searching. Superimposed coding is an alternative that is becoming more attractive as the speed of computers improves.

The superimposed search has a key associated with each record in the data base to act as a screen to eliminate the majority of records from further consideration. The keys are based on the bigrams and trigrams contained in

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

099 700

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

the record, and are arranged in a linear file.

The sequential search is a character by character scan of the entire file. This search is facilitated by constructing a finite state machine at the beginning of the search to match the search terms. The inverted file is fairly standard, except for the use of bit vectors to hold the postings of very common entries.

A data base of 100,000 INSPEC records, from nine months of 1974, was used for testing the algorithms with 339 real-life search questions. The INSPEC file consists of over 120,000,000 characters of which 30,900,000 were in searchable fields.

The principal concern of the study was the computer run time taken to prepare the files for searching and to conduct the searches, although other factors such as clock time, memory requirements and disk I/O were also measured. Detailed analysis of the search algorithms was done to confirm the experimental results and extend these results to searching under different conditions, such as batching search questions and searching larger files.

Searches of the 100,000 records averaged 775 seconds run time for the sequential search, 30 seconds for the superimposed search, and 1.6 seconds for the inverted file. The superior searching efficiency of the inverted file was offset by its increased file preparation time--twice as long as the minimum required to prepare the file for sequential searching. Constructing the linear key index for the superimposed search took only one-sixth the time needed to construct the inverted index. The inverted index occupied 28,416,000 bytes of disk storage, while the superimposed keys occupied 8,000,000.

It was concluded that for files of low to moderate searching activity, the superimposed search is more efficient, in terms of computer run time spent preparing and searching the files, than either the sequential search or the inverted search.

UILU-ENG 77-2208

SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES

by

Thomas Butler Hickey

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES

Thomas Butler Hickey, Ph.D.
Coordinated Science Laboratory and
Graduate School of Library Science
University of Illinois at Urbana-Champaign, 1977

The relative efficiency of three computer search algorithms was compared for searching large bibliographic files with Boolean search strategies. The sequential and inverted files represent the two most common file structures used today for bibliographic searching. Superimposed coding is an alternative that is becoming more attractive as the speed of computers improves.

The superimposed search has a key associated with each record in the data base to act as a screen to eliminate the majority of records from further consideration. The keys are based on the bigrams and trigrams contained in the record, and are arranged in a linear file.

The sequential search is a character by character scan of the entire file. This search is facilitated by constructing a finite state machine at the beginning of the search to match the search terms. The inverted file is fairly standard, except for the use of bit vectors to hold the postings of very common entries.

A data base of 100,000 INSPEC records, from nine months of 1974, was used for testing the algorithms with 339 real-life search questions. The INSPEC file consists of over 120,000,000 characters of which 30,900,000 were in searchable fields.

The principal concern of the study was the computer run time taken to prepare the files for searching and to conduct the searches, although other factors such as clock time, memory requirements and disk I/O were also measured. Detailed analysis of the search algorithms was done to confirm the experimental results and extend these results to searching under different conditions, such as batching search questions and searching larger files.

Searches of the 100,000 records averaged 775 seconds run time for the sequential search, 30 seconds for the superimposed search, and 1.6 seconds for the inverted file. The superior searching efficiency of the inverted file was offset by its increased file preparation time--twice as long as the minimum required to prepare the file for sequential searching. Constructing the linear key index for the superimposed search took only one-sixth the time needed to construct the inverted index. The inverted index occupied 28,416,000 bytes of disk storage, while the superimposed keys occupied 8,000,000.

It was concluded that for files of low to moderate searching activity, the superimposed search is more efficient, in terms of computer run time spent preparing and searching the files, than either the sequential search or the inverted search.

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Martha Williams for her guidance in developing this thesis, especially her insistance that what I believed was impossible could be done, and also for the financial support available through the Information Retrieval Research Laboratory of which she is the director. I would also like to thank my co-advisor Dr. James Divilbiss who first brought superimposed coding to my attention, suggested its use in free text searching, and provided many enlightening discussions about the implemenation. I greatly appreciate Dr. Lucille Wert's encouragement during my course work and her critical review of this thesis.

The Institution of Electrical Engineers generously allowed the use of their INSPEC data base in this study. The search questions were obtained from NERAC at the University of Connecticut through the courtesy of Dr. Daniel U. Wilde, Director, whose cooperation is very much appreciated.

Without the moral and financial support of my wife Carol this study would not have been possible.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

BACKGROUND

Superimposed Coding

Superimposed coding was first developed as a means of encoding document descriptors on edge-notched cards. The advantage of using superimposed coding is that all the descriptors can be entered in the same field on the card, instead of in separate, shorter fields as was once done. Reducing the number of fields reduces the number of spindling operations needed to retrieve documents, making edge-notched cards much easier to use. The same principle, reducing the number of fields which have to be inspected, can be applied to computer algorithms, decreasing the time taken by the computer to perform bibliographic searches.

Calvin Mooers [54] developed and marketed a system that used superimposed coding under the trade name Zato Coding. He obtained a greater coding efficiency, and at the same time decreased the number of spindling operations needed to retrieve a document. In the Zato system the code for each descriptor of a document is entered in the same field, and reliance is placed on the statistical improbability of the superimposed codes producing 'false drops'; records which the system coding says have a certain descriptor, but in fact do not. In any operational system such false drops do occur, but they are normally rare and easy to screen out by hand, especially if the coordination of two or

more descriptors is required. The document files which are indexed by such cards usually contain fewer than 10,000 or 20,000 records.

Many other systems have been developed which use superimposed coding, including some which use 'chain coding.' Chain coding is a step closer to the type of coding used in computer applications of superimposed coding [67], in that it makes use of a table to generate codes for superimposition directly from the index term, thus eliminating the need for the maintenance of a list of the terms and their associated codes.

Moores [54, 55], and several other writers have analyzed false dropping fractions, that fraction of a file that can be expected to incorrectly retrieved. Fraction analysis still seems to generate a good deal of interest. For example Bird [8], in the most recent article, gives a review of the use of superimposed coding in both edge-notched and optical coincidence (or Peek-a-Boo) cards.

The first application of superimposed coding to computer searching appears to have been done by Hutton [36] in his PEEKABIT system. This is a very simple system in which each letter has a specific binary code. Words are overlaid in a field on each other depending on their length, to achieve an evenly distributed code. This technique has the advantage of not requiring a previously assigned code for each descriptor, but it also has the disadvantage of preventing any type of truncation.

Malcolm Harrison [32] was the first to decribe the type of superimposed coding used in this study. In his method, overlapping n-

grams from a string of characters are hashed into a bit field called a key, which can then be searched much more quickly than the original string of characters.

Figure 1 is an example of superimposed coding using single characters (1-grams) to encode ten authors of ten fictitious documents. There are two ways of looking at this encoding. The first corresponds to 'terms on postings', where in this case the terms are simply letters. From this viewpoint we have ten keys each representing the author of one document, each bit in the key noting the presence or absence of a character in the author's name. To check whether an author is contained in the file, each of these keys has to be inspected in turn. In a computer such an inspection could be done in a single operation for each key.

The second way of viewing this encoding is the inversion of the first way, looking at it as 26 columns, one column for each character. This is the 'postings on terms' orientation. Each column contains as many bits as document authors, in this case only ten. The bits indicate whether the document they correspond to has an author whose name includes the corresponding character. Searching proceeds by reading in each of the columns corresponding to the characters wanted in the search term, and checking for a document that has all the characters required. This coding is too simple to use in an actual system if the descriptors for a document are much longer than the authors in the example, since more descriptors would result in a high proportion of all the letters

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
01: ALBRIGHT,JOHN	X	X					X	X	X	X		X	X				X	X								
02: AULD, LARRY	X		X									X					X		X						X
03: BERNAL-ROSA, EMILIA	X	X			X				X			X	X	X	X		X	X								
04: BLUE, RICHARD	X	X	X	X	X	X				X	X						X		X							
05: DAVIS, NATHANIEL	X		X	X					X	X		X		X												
06: HALL, RICHARD	X	X							X	X		X					X									
07: HICKEY, THOMAS	X		X		X		X		X	X	X	X		X												X
08: KRIEGER, TILLIE				X		X		X	X	X		X					X	X								
09: LUCAS, LINDA	X	X				X			X	X		X		X					X							
10: OMALLEY, KENNETH	X			X					X			X														X

'Terms on Posting'

	01	02	03	04	05	06	07	08	09	10
A	X	X	X	X	X	X	X		X	X
B	X		X	X						
C				X		X	X		X	
D		X		X	X	X			X	
E			X	X	X		X	X		X
F										
G	X							X		
H	X			X	X	X	X			
I	X		X	X	X	X	X	X	X	
J	X									
K							X	X		X
L	X	X	X	X	X	X		X	X	X
M			X				X			X
N	X		X		X				X	X
O	X		X				X			X
P										
O										
R	X	X	X	X		X		X		
S			X		X		X		X	
T	X				X		X	X		X
U		X		X					X	
V					X					
W										
X										
Y		X					X			X

'Postings on Term'

Simple Superimposed Code

Figure 1

occurring in all of the documents. To enable more descriptors to be used, a longer field must be provided, and some quite complex methods of encoding have been developed both for punched cards and computer applications.

More recently Barton et al. [7] have described a technique which uses superimposed codes derived from equifrequent substrings. This technique has the advantage of giving a fairly even distribution of bits in the keys, but as this type of key is expanded to lower the false drop fraction, the longer keys tend to become less and less dense, a disadvantage. (See page 59 for a discussion of entropy and its relation to keys.) Barton's implementation has an organization analogous to 'postings on terms.' For each substring chosen, a string of bits (called a 'bit vector') is generated, each bit of which indicates whether the source document contains that substring. The main disadvantage of using equifrequent substrings is the increased amount of processing time required to identify the substrings' occurances within a record.

Keys of the type Harrison proposed have received more attention in the literature than Barton's. Bookstein has proposed using such a key as a secondary screening device after retrieving a group of records via an Ohio College Library Center (OCLC) type hashed search key [11]. Donald Knuth, in an investigation of fast linear searching mentions Harrison's technique as a type of search potentially faster than the one he and his colleagues have developed [42, p. 31]. Goble has used a

variation of this type of key, based on 4-grams, in an SDI system. In Goble's system each record is read in, a long key is created containing pointers to the positions of the 4-grams in the record. This is really an inversion technique on single records [31].

This author first became aware of superimposed coding from the work of Ehrhardt [26]. Ehrhardt uses superimposed coding for a selective dissemination of information (SDI) and retrospective search service. In this system there is a key for each word in the index, the keys based on single letters. Ehrhardt's method permits truncation in a novel way; a Key Letter In Context (KLIC) index is kept of all the commonly occurring index terms in the data base. By searching the KLIC, all words which contain a word fragment can be identified and entered into the search system as full words. He reports a decrease in run time of a factor of ten over his previous search system which did a character by character match.

Superimposed coding is also used in chemical substructure searching of some very large files [27], but the techniques needed in these circumstances are very different from those needed for searching text strings, such as occur in bibliographic records. Lefkovitz has pointed out a potential of superimposed coding [45]. He notes that the increased speed of the newer computers makes the technique more useable for large files.

Current Approaches to Subject Searching

The two most common file organizations used for subject retrieval are linear and inverted files [43, 44]. A linear file is the simplest of all organizations, in which the records are simply in a list and the whole file has to be searched character by character in order to retrieve the records on a subject. Although the linear file search speed is slow compared to other types of search, this file organization is used a great deal by SDI services. Since indexes do not have to be prepared to begin searching a linear file, the result is a minimum delay in searching new files. Computer time is minimized because searches for many different people can be done simultaneously, reducing the cost per search, and, since SDI is a current awareness service, the tapes are expected to be searched only once. Retrospective searches on linear files are occasionally undertaken, but are quite expensive. The ease with which a very simple linear search can be programmed is also often a deciding factor in its selection.

Inverted files are used for most retrospective searches. Through a series of indexes (see Figure 2) rapid access can be made to any record in the file, resulting in a much shorter response time than is possible with linear files. Online search systems such as ELHILL and Lockheed's DIALOG invariably use variations of this file structure. The major disadvantage of inverted files is that the index to the file is itself a large and complex file, often approaching the original file in size. To add new records to such a file, each searchable term in each

Index Entry	Postings
VARIATION	56398
VELOCITY	56405, 56407, 56408
VELOCITY MEASUREMENT	56403
VELOCITY MEASUREMENTS	56405
VELOCITY OF SOUND	56406
VIBRATIONS	56401
VIBRATOR	56402
WATER	56405, 56407
WAVE LENGTHS	56401
WAVE ULTRASONIC VELOCITY MEASUREMENTS	56405
WAVE VIBRATOR	56401

Each number in the postings list is a link to the complete reference in which the index entry occurs.

Typical Inverted File Organization

Figure 2

new record has to be entered into its proper place in the index, often forcing a complete reorganization of the index to make room for the new entries. This is expensive in computer time and requires much more complex software for file maintenance than a simple linear file for which new records are simply added to the end of the file.

This thesis is an attempt to compare superimposed coded files to linear and inverted files, mainly in terms of the CPU (Central Processing Unit) time taken to prepare the files and perform searches of them. (See page 12 for a formal statement of the hypothesis.)

Methods of File Structure Analysis

Many factors influence the decision on how a file is best or most efficiently organized for searching [16, 47]. For example, file size, file stability, required response time, arrival rate of search questions, possibility of batching searches, storage requirements, programming complexity, CPU time required to perform the searches, and the types of search questions expected should all be considered.

There are two basic approaches to evaluating file structures: modeling and implementation. Mathematical analysis of the file structure is the basis of modeling. Costs in terms of run time and other factors are often obtained using simulation programs based on such analysis. Analysis and simulation offer the possibility of exploring many approaches with a minimum expenditure of computer time.

The principle problems with this approach are that the

simplifying assumptions needed to make the analysis possible may make the results suspect, and that truly general purpose simulation programs do not exist. Although some programs, such as Cardenas's [16], can handle several types of file structure and file characteristics, any simulation program is limited in the variety of structures testable by it. Idiosyncrasies of the files, and of user requirements, both of which may have an effect on the performace of the search programs, are impossible to predict.

In particular, I have not found reference to any generalized simulation systems capable of handling free text bibliographic files. Most of the simulation work done in this area is concerned with management information systems (MIS) which work with files very different than bibliographic data bases. The more complex simulations begin to approach actual implemenations as they take more and more into account. File construction costs are very difficult to estimate, and most simulations simply ignore this area [17].

Implementation is not really an alternative to modeling, as at least some analysis of what a search algorithm is to accomplish is needed for a successful implementation. Timing and analysis of any program is vital if there is to be efficient operation, as the slowest sections of programs are not always where prior analysis has predicted. In a study such as this where several file structures are considered, the important advantages of actually implementing the searches are that the complex interactions of the queries, data base and file structure

are not ignored, and that working searches based on a novel file structures, insures and demonstrates that the analysis phase has not omitted or glossed over any critical objections to its feasibility.

CHAPTER II

METHODOLOGY

Hypothesis

The guiding hypothesis of this thesis is: There is a range of number of searches for which the cost of file preparation plus the cost of the individual searches will be less if superimposed coding is used than if either a linear or inverted file structure is used.

This relationship between the the number of searches and cost was expected to occur because the cost of preparing a file for a superimposed search is only slightly more than the cost of preparing a file for linear searching. In addition, the searching efficiency of superimposed coding is much greater than that of a character by character linear search.

Cost was restricted to computer (CPU) machine time. This was done to simplify the study on the assumption that CPU time is the critical factor in assessing the efficiency of these three file structures. Other factors such as memory requirements (both on disk and in core), amount of I/O (input-output to disk), and clock time (related to response time) were also measured and will be discussed, however optimization of these factors was not a major goal in the design and implementaion of the different file structures.

A search consists of determining the presence of the terms of a search question in a record and evaluating the search question's logic.

Search questions used consisted of from one to 46 terms and their associated Boolean logic.

Research Design

The basic approach used was to actually implement the three search algorithms (linear, superimposed and inverted) on a file large enough to be representative of the data base as a whole, and large enough to present the type of problems encountered in implementing large operational systems. The timings of the computation involved in constructing especially the inverted file are complex and a simulation based on a small sample of the file would not be accurate. Implementation also demonstrates the feasability of constructing a superimposed file on the basis of the n-gram structure of a small section of the file.

The INSPEC (Information Service in Physics, Electrotechnology and Control) data bases was selected (see page 16) which is in many ways typical of the types of data and problems of bibliographic files on which linear and inverted searches are currently being done. A large number of search questions (339) specifically designed for this data base was obtained. The entire batch of search questions was run against the inverted file only, because of the great amount of computer time needed by the superimposed and linear searches. To ensure that as representative a sample as possible was used for these two searches, the questions were stratified on the basis of two criteria: number of terms

in the question, and number of records in the file which satisfy the question. This information was obtained by running all the questions against the inverted file. These two criteria were selected because they are the most likely to affect the speed of most search algorithms. Another potential criteria considered was the number of synonym groups in each question, but this was not expected to have as strong an effect on the search timings as the number of terms in a question.

The file was split into three sections for each of these criteria, each of the sections with as close to one-third of the questions as was possible. This divided the 339 searches into nine approximately even groups. For the linear search nine or ten searches was the maximum that could be run against the whole file, because of the great amount of CPU time taken by each search, so one question from each of the groups was selected with the help of a pseudo-random number generator. This technique was also used to select questions for the superimposed search, with the added need to weight the sample according to the number of questions in each stratum.

The same algorithm is used for the linear search as is used by the superimposed search for its secondary scan of the records to screen out false drops. Both these searches retrieve exactly the same records. The inverted search is unable to retrieve exactly the same records found by the linear and superimposed searches because the very nature of the inverted index restricts the search to terms as entered in the index, with only right truncation allowed. The lack of left truncation has an

effect, even if only full words are used in the search questions, since phrases may occur in the inverted index. The technique of breaking the free index phrases at each word has been used to reduce this effect, and in the actual searches there are only a few search questions which retrieve different numbers of references under the different searches. The differences were small and did not influence the timings. An alternative would have been to restrict the inverted file to single words, but this would have been an unfortunate decision since the questions (received after the programs had been written and the files generated) make extensive use of phrases.

In the design of all of the programs, when there was a choice, speed of execution took precedence over memory size, both in core and on disk. The core memory available is more than adequate for these programs, but disk space is more of a constraint. SAIL (Stanford Artificial Intelligence Language), a high level language, was used (see page 20) for the programs, with a few small (no more than five or six lines of SAIL each) sections of the translation program and searches rewritten into assembly language.

Timings of sections of the programs were done all through their development to optimize them as much as possible. Detailed simulation of the searching efficiency of various superimposed keys was done to decide on the exact configuration of the superimposed file. Analysis and timings were done on the completed search programs both to compare each search's performance on this data base, and to provide a basis for

extending the results to other and larger data bases. Measurements of other computer costs such as clock time and disk I/O were also measured and compared with the results based on run time.

Database

The Institution of Electrical Engineers (IEE) generously extended the period in which IRRL was allowed to use their INSPEC data base to test these file structures.

The INSPEC tape service is based on the data used by the IEE to produce Science Abstracts. INSPEC covers the subject areas of electrical engineering and elctronics, physics, and computer and control engineering. Files are distributed twice a month on magnetic tape with references to all three of the abstracting services intermixed. It primarily indexes journal articles, but also includes many conference proceedings, books, technical reports and patents. The advantages of using this data base were:

1) The large number of records available. IRRL has more than 150,000 records from the period January 1974 through March 1975.
2) The variety of indexable fields. These fields include authors, editors, controlled and free vocabulary (single index terms and phrases), CODEN, and subject codes.
3) The great depth of indexing. The indexable fields of each record averaged more than 300 characters, making the data base a severe test of superimposed coding.

4) INSPEC is one of the most widely distributed and searched data bases by search services, most of which use linear or inverted file structures.

5) INSPEC's coverage is similar to many other data bases searched in this manner. Most bibliographic data bases emphasize the journal and technical report literature, as INSPEC does.

Two other data bases, also available within IRRL, were used in preliminary investigations. The MARC (Machine Readable Cataloging) data base from the Library of Congress (LC) was not used because book cataloging is not typical of most of the subject searching done by computer today. The other data base available, Science Citation Index published by the Institute for Scientific Information was used extensively in preliminary studies of superimposed coding, but was not used in the final study because of the limitations in its indexable fields--only authors and titles are available. Either of these data bases would actually have been easier to implement for superimposed coding than INSPEC, but would not have been as severe a test of the wider applicability of superimposed coding.

A total of 100,000 records was used to test the different file structures. This covers a period of slightly more than nine months and was selected because this was approximately the largest number of records that can be handled, within the limitation of available disk space. A smaller file would have been enough to obtain accurate timings on the linear and superimposed searches, but the inverted file

is dependant on vocabulary size. A file of nearly this size is needed for it to be representative of the vocabulary of the complete file, and to make possible accurate determination of the cost of building the inverted files.

No single bibliographic data base can represent the qualities of all other data bases, since it is well known that specific data bases have specific characteristics that are different from other data bases. INSPEC can be considered typical of most 'bibliographic' data bases--those dealing with references to journal articles, reports and books. Data bases of other types, such as those containing numeric or chemical structure data, are very different in their makeup and type of queries that they must handle. Testing superimposed coding on INSPEC gives little information about the applicability of superimposed coding on non-bibliographic data bases.

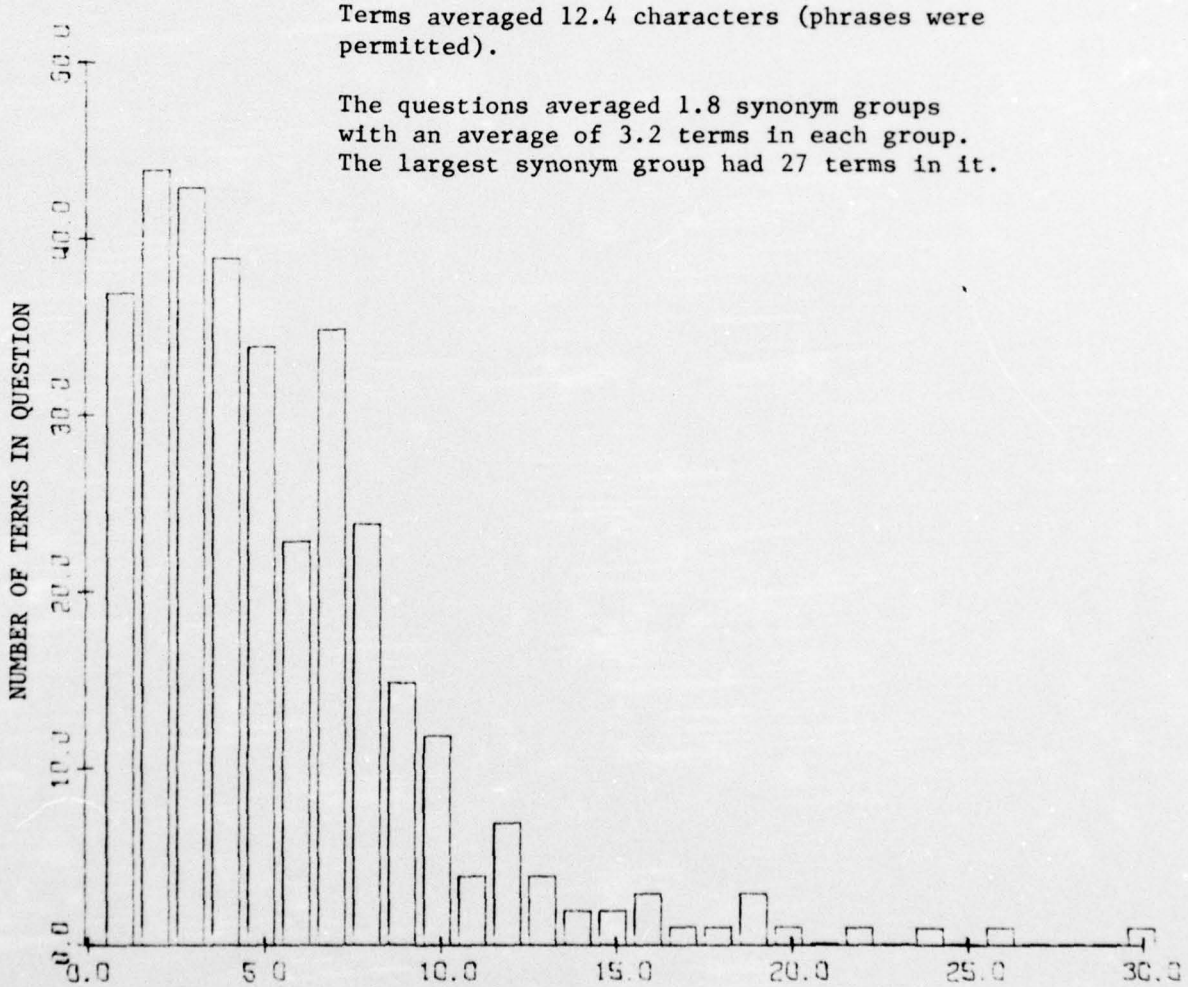
Search Questions

Almost as important as the data base to be searched are the questions which are used to test the file structures. The New England Research Applications Center (NERAC) at the University of Connecticut generously supplied a large number of search questions specifically designed and used by them to search INSPEC in an offline situation for SDI. Figure 3 is a summary of the questions' characteristics. These questions are from actual searches run by NERAC in their current awareness service, and represent a range of topics and search

There were 339 search questions with a total of 1956 terms, an average of 5.8 terms per question. 23 of these terms were negated.

Terms averaged 12.4 characters (phrases were permitted).

The questions averaged 1.8 synonym groups with an average of 3.2 terms in each group. The largest synonym group had 27 terms in it.



Search Question Characteristics

Figure 3

formulations that would have been impossible to achieve with a synthetic set of search questions. (See page 13 for a description of how samples of the these questions were selected for searching.)

The NERAC queries provide a good indication of the types of queries used in the batch environment to search bibliographic data bases. It is an advantage that they were developed specifically for searching INSPEC, since to be successful all searches have to reflect the content of the data base being searched. The number of hits the questions produce may be similar to that found by online searches, but their form is very different since online searches are usually built up piece by piece--first searching a very broad subject and then narrowing the references to a manageable number by adding constraints to the search. The NERAC search strategies have already gone through this interaction over a period of months.

Implementation Language

The preliminary programs were written in BLISS, a systems programming language for the DECsystem-10. BLISS has the advantage of compiling very efficient programs, and the disadvantages of being difficult to write, lacking such features as built in I/O routines, and that it is not widely used. All of the programs used for timings in this study were written in SAIL, which is an extension of ALGOL-60 for Digital Equipment Corporation's DECsystem-10 computer. In addition to being much closer to ALGOL than BLISS is, SAIL has routines that give

full access to the I/O capabilities of the DECsystem-10, extensive string manipulation capabilities, and the feature of being able to code blocks of programs in assembly language.

By coding small parts of programs in assembly language, large gains (sometimes a factor of two or three) are often possible in computer programs. This was done in the programs written for this study whenever timings of the program showed that the gain in execution speed would be worth the extra effort involved in recoding the program section, and the loss in ease of modification.

Computer Facility

The computer used in this study was a DECsystem-KI10 manufactured by the Digital Equipment Company, a computer very popular with scientific research laboratories. It is a fairly modern third generation computer (Digital now makes a DECsystem-KL10 which is somewhat faster) which uses a 36-bit word and seven-bit ASCII characters internally. It is a computer designed for timesharing; running in batch mode offers no advantages over timesharing except exemption from terminal connect time charges.

Removable RP03 (IBM 2314 dual density equivalent) disks, holding 51,200,000 characters each, were used for temporary and permanent storage of the data base. I/O to disk on the DECsystem-10 is done in units of 128 36-bit words (640 characters) and can be done either in a buffered mode allowing concurrent I/O and computation, or in dump mode

which allows large blocks of data to be transferred at one time. Direct access can be done with either of these modes.

The main memory is magnetic core. More than the maximum address space of 256 K was available if needed, but all of the programs in this study ran within the normal maximum of 70 K. The KI10 has 16 registers available to each program, but there is no cache or intermediate memory between core and the CPU. The operating system used was the standard monitor supplied by the manufacturer, 'TOPS' version 6.02. This is among the most sophisticated operating systems available, and allows priorities to be set on CPU and disk usage. All of the programs in this study ran under normal priority, but were often run late at night to take advantage of lower rates. During late night runs there is very little competition for the system resources.

One of the best features of the DECsystem-10 is the ease of timing. The system clock is readily available, and gives run time and clock time (sometimes called 'real,' 'day' or 'wall clock' time) to the 60th of a second. Nearly all the system overhead associated with a job, such as is involved in I/O, is charged directly to the job by the run time it takes, so that run time is an excellent indicator of the cost of a program[1]. Run times are very consistent over several runs of the same program, especially when system activity is low, and the timings appear to be accurate to within a few percent.

---

[1]This is not true on many computers, notably IBM's 360/370 series of computers in which I/O is done in a supervisor state and charged separately to the user.

CHAPTER III

PROGRAM DESCRIPTIONS

Data Base Translation Program--DINTRN

The INSPEC data base is distributed in a format that complies, as does LC's MARC, with the American National Standards Institute standard for bibliographic information interchange on magnetic tape [6]. Tapes are written in Binary Coded Decimal Interchange Code (BCDIC), an eight bit character code. (See Appendix A for a listing and definitions of the fields which can occur in the INSPEC records.)

Although similar to the MARC format, INSPEC has some properties which make it very easy to work with. None of the fields occurs more than once in a record, the fields are always in numeric order, and each field 001 contains a unique identification number assigned sequentially to the records.

The following fields were selected to be searchable:

120	Sectional Classification Codes
121	Unified Classification Codes
130	Subject Index Headings
131	Free-Indexing Terms
200	Author(s)
210	Editor(s)
310	CODEN
311	CODEN of Cover-to-Cover Translation

The title is not used to generate index terms since the free index terms include virtually all the important terms in the title, plus many that only occur in the abstract [4, 5].

The size of the records posed a problem. With a raw record length of nearly 1,300 characters the anticipated 100,000 record data base would occupy 130,000,000 characters of storage. Since the mountable disk packs available for use on CSL's DECsystem-10 will hold a maximum of 51,200,000 characters, some sort of compression and/or truncation of the records was inevitable.

The abstract is the largest field on the records, averaging more than 500 characters, so these abstract strings are separated from the rest of the record and written into a separate file. The fields to be searched averaged about 320 characters per record (62 percent of a disk for 100,000 records) and the rest of the reference after some formatting averaged about 260 characters (50 percent of one disk unit). It is desireable to minimize the amount of data read in, both for the linear search which has to read every record into core for searching, and for the file processing programs which build the indexes to the data base for the other searches. Since the combination of the fields to be searched plus the references is too large to fit on a single disk, and because of the desireability of keeping the portions of the record to be searched separate from the rest of the record, these are split into two files. The searchable fields are concatenated to form the index string and the rest of the reference minus the abstract, goes into a reference

string. As the records are translated each of these three strings (the abstract, reference and index strings), was written into a separate file, three files for each tape processed. Figure 4 is an example of an INSPEC record and the strings it was divided into.

The only directory kept in these strings is a 36-bit leader giving the string length in characters to make it possible to use binary I/O, much faster on the DECsystem-10 than character by character I/O. With binary input, first the length of a string of characters is read in and then the string is read in (as if it were 36-bit binary numbers) into an array which is then treated as character information. Character input is much slower because each character is separately moved and checked for an end of string marker. The reference string is formatted so that it is approximately ready to display as is, and the index string is made up of the searchable elements preceeded by their tags so that searches can be restricted to specific fields. The tags used are:

S: Sectional Class Code

U: Unified Class Code

C: Controlled Subject Index Heading

F: Free Index Term

A: Author

E: Editor

J: CODEN

Statistics on character and word counts and frequency of occurence are kept on all the INSPEC fields; these are printed for each tape and

Original Record With Directory

00820NA>>>1100241>>>45>>00100090000001000050000910
00064000141100123000781200020002011210016002211300
06100237131012400298132000400422150002800426200001
60045430000200047031000090049040000150049962000100
05146300004005247000035005288100013005631\1$585039\
1$02\1$A NOTE ON OPTIMAL APPROXIMATING MANIFOLDS O
F A FUNCTION CLASS\1$EXPLICIT EXPRESSIONS FOR N-WI
DTH AND EXTREMAL SUBSPACES ARE OBTAINED FOR A CLAS
S WHICH IS OF SOME ENGINEERING IMPORTANCE\2$B1410$
C6210$C8230\1$DLCAAC$DLTGAM\3$LINEAR ALGEBRA$APPRO
XIMATION THEORY$FUNCTION APPROXIMATION\1$N-WIDTH S
UBSPACES$EXTREMAL SUBSPACES$LINEAR ALGEBRA$FUNCTIO
N APPROXIMATION$OPTIMAL APPROXIMATING MANIFOLDS$FU
NCTION CLASS\1$T\1$BELL SYST. TECH. J. (USA)\1$NET
RAVALI, A.\2$B7400034$C7400002\1$ESTJAN\1$VOL.52,
NO.7\1$1237-42\1$6\1$BELL SYSTEMS LAB., NEW YORK,
USA\1$SEPT. 1973\!!!

Reference String

INSPEC # 585039,B7400034,C7400002
THEOR/MATH, JOUR PAPER
TITLE: A NOTE ON OPTIMAL APPROXIMATING MANIFOLDS
OF A FUNCTION CLASS
BELL SYST. TECH. J. (USA) VOL.52, NO.7 P. 1237-42, SEPT. 1973
REFS: 6
AUTH AFFIL: BELL SYSTEMS LAB., NEW YORK, US

Index String

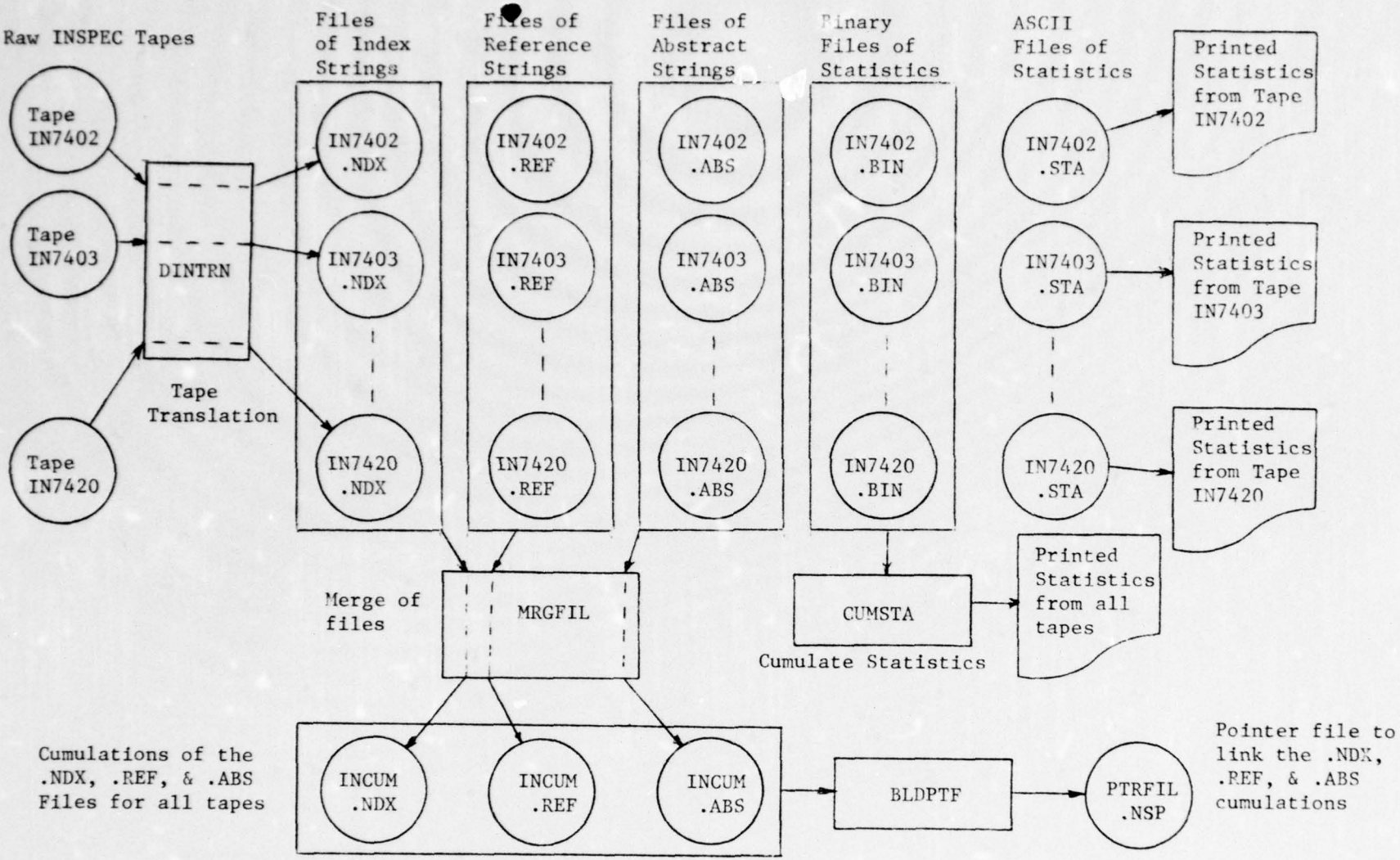
585039 S:B1410 S:C6210 S:C8230
U:DLCAAC U:DLTGAM
C:LINEAR ALGEBRA C:APPROXIMATION THEORY
C:FUNCTION APPROXIMATION
F:N-WIDTH SUBSPACES F:EXTREMAL SUBSPACES
F:LINEAR ALGEBRA F:FUNCTION APPROXIMATION
F:OPTIMAL APPROXIMATING MANIFOLDS F:FUNCTION CLASS
A:NETRAVALI, A.
J:BSTJAN

Abstract String

585039
EXPLICIT EXPRESSIONS FOR N-WIDTH AND EXTREMAL SUBSPACES
ARE OBTAINED FOR A CLASS WHICH IS OF SOME ENGINEERING
IMPORTANCE

Sample INSPEC Record

Figure 4



Tape Translation Flowchart

Figure 5

also stored as a binary file for cumulation. Figure 5 is a flowchart showing how the tapes are named and processed. To improve the run times of these programs most string handling is done by specially written assembly language routines, as is the translation from eight-bit BCDIC to seven-bit ASCII characters.

Routines Shared by All Searches

There are several procedures common to all three searches. Many of them are of little direct interest to this study because they provide support functions such as displaying previous commands, combining previous searches, guidance in response to a request for help, and display of search results on the screen. These functions can be used at the discretion of the searcher, but were not used when running the tests with the fully developed search questions available. See Appendix B for detailed descriptions of the interactive use of the programs.

When the search programs are used interactively the results of each search are stored in a bit vector 100,000 bits long. Each bit in this vector indicates whether the corresponding record in the file satisfied the search question. The search system assigns a unique number to each search and previous searches can then be combined by specifying the Boolean logic in terms of these numbers. Internally this logic is transformed into reverse Polish notation with no restriction on the logic. Because the searches are represented by bit vectors, such combination can be done very quickly with any number of postings (up to the maximum possible of 100,000).

Each bit vector occupies nearly 3,000 36-bit words of main memory, and therefore only a limited number can be kept in core concurrently. The remaining bit vectors are stored in a direct access file on disk. A bit vector handler routine was written to keep track of the location of bit vectors; if the search requires a bit vector that is not in core, the handler swaps the least recently used bit vector onto disk, and reads the new one into core.

The system works very well with the swapping, essentially a simple virtual memory system, barely noticeable to the user. The maximum number of bit vectors allowed depends on the disk space available, and the number in core at one time depends on the core storage available. Both can be specified at compile time, with a present limit of five searches' bit vectors in core, and a maximum maintained on disk of 40.

As will be explained in the section on the operation of the inverted search, although ORing the postings together for a search is normally handled within the index search routine, ANDing and NOTing the bit vectors is done with the procedures in the bit vector logic module, BVLOG. Module BVLOG provides procedures to complement, AND, OR, and clear bit vectors; to convert a list of postings into a bit vector; to convert a bit vector into a list of postings; and to count the number of postings held by a bit vector. These routines are called by several sections of the search programs.

Several procedures are used by all the searches to process the query prior to searching. After a query is read in, the first step is

to verify that the query is in a Boolean form acceptable to the parser. This routine uses a finite state machine which makes a state transition for each parenthesis, Boolean operator and term in the search question. To accomplish this a table is kept of the acceptable state transitions, e.g. after an OR operator a NOT operator is an illegal input to the state reached after checking the OR operator. At this point an error message is sent to the terminal with an indicator pointing to the location of the error. Run time of the logic checker is linearly dependant on the length and complexity of the search question, and is on the order of 0.005 seconds for a typical search.

The next step the search question undergoes is the truncation procedure, TRUNC, which modifies the queries by removing all parentheses and setting spacing to indicate truncation modes to the searches. Past this point the truncation mode of the search term is implicit in whether the term is proceeded or trailed by a delimiter (a blank). The term 'ELECTRIC' will match 'ELECTRICAL', 'ELECTRICITY', etc., but the term 'ELECTRICØ' would not[1]. 'ELECTRICØ' will match either the term 'ELECTRICØ', or a phrase such as 'ELECTRICØMOTORSØANDØMACHINERY'. Care is taken in the generation of the inverted file to ensure that a blank followed each term in the index, and that delimiters occurred on both sides of terms for the superimposed and linear files.

Procedure TRUNC has not been optimized at all and runs fairly slowly, mainly because it forces the SAIL run time system to do several

---

[1]The Ø represents the character space.

string space garbage collections. Its run time is between 0.02 and 0.05 seconds, with an average of about 0.03. It is proportional to the length of the search question and the number of modifications that need to be done on it before it is ready to send to the parser.

Considerable simplification of the query parser is possible because of the restriction of questions to a product of sums form. A typical question in this form is:

[CAPACITANCE#+CAPACITOR#]

* [DIODE#+JUNCTION#] * SILICON

where the '#' sign signifies truncation, the '+' means OR and the '*' AND. Product of sums is a canonical form into which any Boolean query can be transformed.

Out of the more than 300 search questions received for this study, 15 needed modification to fit into this form. A typical example of this is a question in the form:

A+[B+C]*D

To transform this question the term A needs to be repeated:

[A+B+C]*[A+D]

This is logically the same as the original, and will retrieve the same records as the original in any search system. An actual example of this from the search questions:

[GLASS-METAL SEALS + BGGEAR]

* [ALUMINUM + COPPER + GLASS-METAL SEALS]

where BGGEAR is INSPEC'S Unified Classification code for the general area of 'Pressure Measurements and Techniques in Vacuum.'

This type of restriction has also been present in other information retrieval systems, such as CARUSO [43 pp. 528-587]. When used online, each search is automatically given a number, and the numbered searches can be combined in any Boolean combination by the searcher with no restrictions on Boolean form.

The restriction to product of sums form has other advantages, in addition to simplified parsing. When doing an inverted search, instead of building bit vectors for each of the terms A, B, and C in the example, single bit vector is built by simply successively inserting the postings for each term into the same bit vector. This is a very efficient means of merging these lists of postings, and is made possible by the product of sums form of the search.

The superimposed search also takes advantage of the restriction to product of sums when it tranforms the search into a sum of products form, the other Boolean canonical form. The expression

A*A + B*A + C*A + A*D + B*D + C*D

is the sum of products form for the example above. This transformation is fairly easy to accomplish if it is known that the query is in product of sums form, but quite difficult otherwise. In the linear search the logic evaluation can terminate if it is false as soon as it determines that any synonym group is not represented in the record, saving execution time.

Another advantage of the product of sums form, for the purpose of this study, is that it forces a clearer definition of what a search

question is. For a question to be easily represented in this form it normally involves a search on a single concept. The following is one of the questions received from NERAC which were broken into two separate searches:

CRYSTAL FILTER * [AUDIO + ACOUSTIC]

+ [DIGITAL STORAGE + READ ONLY STORAGE + XKAAD]

* [METAL-INSULATOR-SEMICOND# + SEMICONDUCTOR#]

This question contains two distinct concepts (the first about crystal filters, and the second about digital storage), and is better searched as two questions. Two of the search requests received from NERAC were split; both into two search questions.

The main disadvantages of this restriction are that there are some valid search questions which require more work to enter in this format, and some searches may have slightly longer run times because of the repeated terms. The parser used is quite fast, requiring only about 0.01 seconds CPU time per search.

After parsing, the queries are stored as a string array with one array element for each term. Associated with this is an integer array with as many entries as synonym groups in the query, each of which keeps track of which term is the last term in that group. There is also a Boolean array which notes for each term whether the term is to be negated or not.

Inverted File Search--INVSER

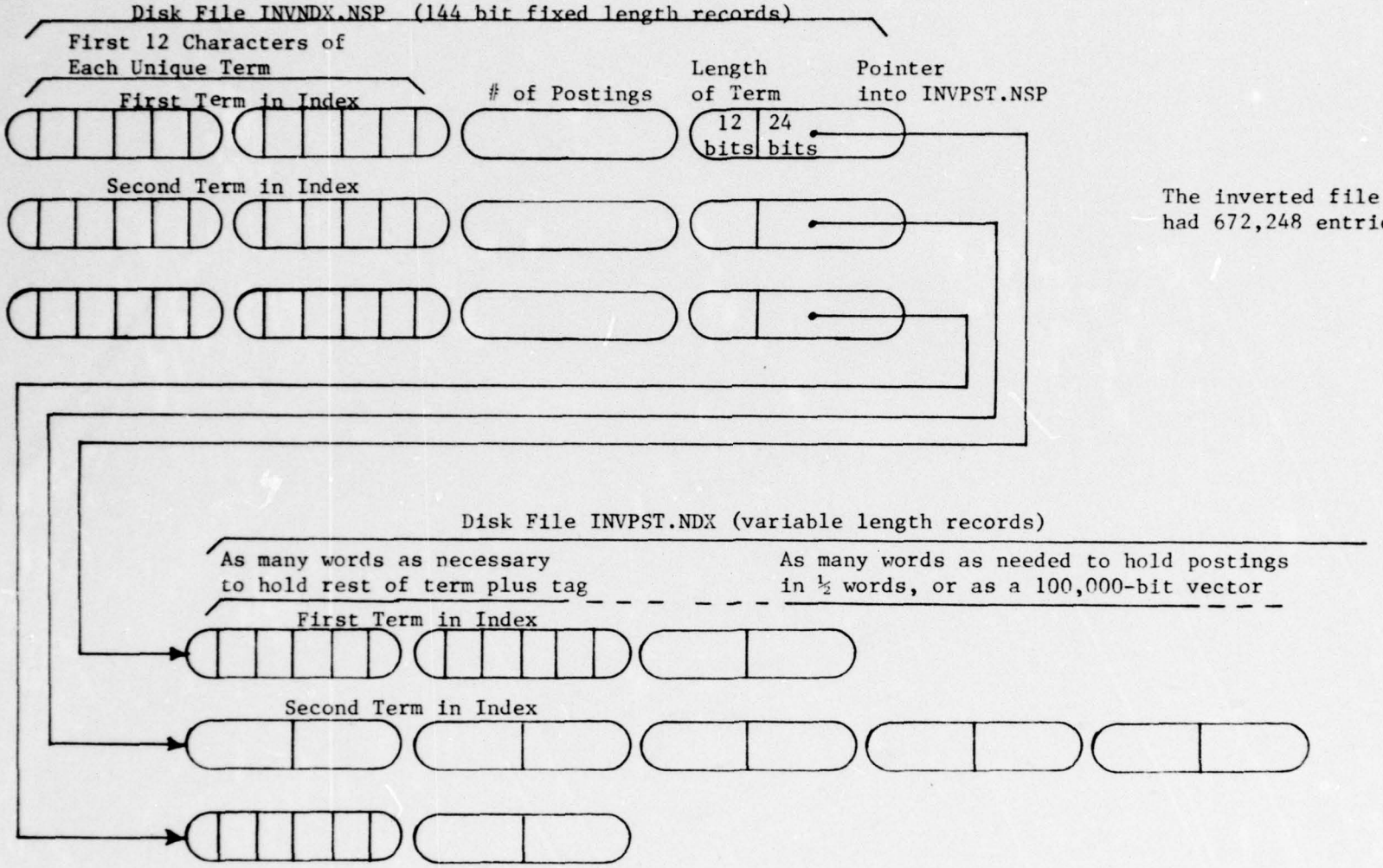
The inverted file is the most complicated data structure on disk used by any of the searches, and its exact form can have a large effect on its performance. The goals set for the inverted file search were:

1) Speed of file generation. This is felt to be of utmost importance because this is the area in which the inverted file is at a disadvantage with the other structures.
2) Compatability with other searches. The other searches are more flexible because they allow left truncation, however by entering the free index phrases under each word in the phrase, as described on page 40, the records retrieved by the inverted search were nearly identical to those retrieved by the linear and superimposed searches.
3) Search speed. This is gained by having two levels of indexes to the inverted file (including a very large and detailed one) in core, and fixed length records holding the first 12 characters of each term. Disk input is minimized for searches with a large number of matches in the inverted index because the entries in the continuation file are also contiguous, eliminating the need to do direct access input for every entry. This speeded some searches by a factor of five, and is possible because the file is built all at once.

The index on disk is divided into two files. The first file (INVNDX.NSP, for Inverted index--INSPEC) contains a fixed length, four-

word (144-bit) record for each unique term encounterd in the data base, sorted into alphabetical order. The first two words of the record are divided into 12 sixbit characters containing the first 12 characters of the indexed term. 'Sixbit' is a character code used by the DECsystem-10 computer to save space. The code is based on seven-bit ASCII and allows six characters per 36-bit computer word instead of the five possible with the ASCII code. The next word contains the number of postings for this term. The fourth word contains two fields, the first 12 bits hold the length of the term in characters (for a maximum length of 4,095 characters), and the last 24 bits are a pointer into the disk file INVPST.NSP (Inverted Postings-INSPEC). Figure 6 shows the structure kept on disk for the inverted file.

The file INVPST.NSP contains the variable length information associated with each indexed term. First it has as many words as are necessary to hold the rest of the term from character 13 on, six characters per computer word. Next follows a list of postings as binary numbers in half words (18 bits), or if the number of postings exceeds 2000, the postings are kept as a bit vector, each bit representing the presence or absence of that term in its associated record. If there are 100,000 records in the data base it is more compact to represent the postings in a bit vector if a term has more than 5,558 postings, since 100,000 bits is the same length as 5,558 18-bit numbers. Allowing two formats for the postings conserves space, and makes it possible to not set any upper limit on the number of postings a term may have associated with it.



Disk Data Structure, Inverted Search

Figure 6

This file is not easily updated, i.e. additions and deletions of records are not possible without major modifications of the whole file. It is typical of the type of data base in this study that such updates are done only periodically (once a week or month), with large numbers of changes for which major modifications of the file would have to be done in any case, and it would not normally be done incrementally. Generalized file maintenance procedures would have been impossible to program in the time available, and a file structure built for easy modification would have been at least slightly slower to search and construct than the one used. In a system which allows incremental updating, additions to the file could be expected to take at least as long per record as was taken in constructing this inverted file. Updating inverted files is always a problem in their design and maintenance, a problem that the superimposed and linear structures avoid almost completely.

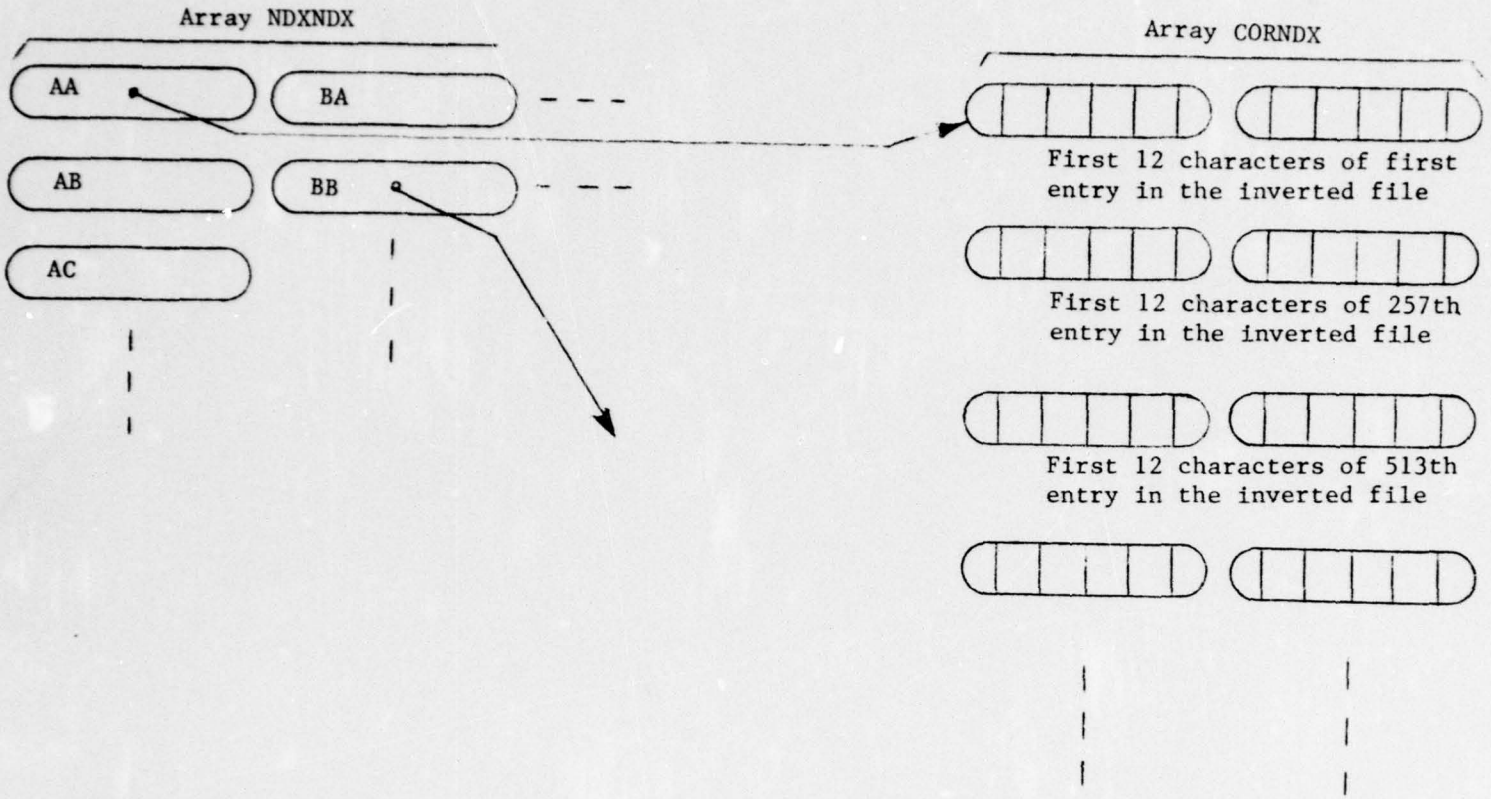
Because of the great number of entries in the file INVNDX.NSP (more than 650,000), two levels of indexes are kept in core. The first is called CORNDX (Core Index) and contains the first two words of every 256th entry in INVNDX.NSP. Originally this index was to keep track of the first six characters of every 32nd term, but it was found that the first six characters were not specific enough, and that the size of the index (greater than 20K) was too large to be comfortably held in core. Above the CORNDX there is a last level of index, a two dimensional array, NDXNDX (Index to Core Index) accessed via the first two

characters of the search term to give the first location in CORNDX which starts with those two characters. Figure 7 shows the data structure of indexes kept in core to the inverted file.

After the question has been parsed, procedure QSER (Question Searcher) handles the search, calling procedure INVSER (Inverted Searcher) with each term in turn. Procedure INVSER actually searches the index. First the search term is checked to see whether a tag is present and an array of masks is set up to facilitate comparisons of the strings in sixbit characters. The term is then converted into sixbit from the normal seven-bit ASCII code. An entry to CORNDX is found via NDXNDX on the basis of the first two characters of the search term, and CORNDX is linearly searched from this point to find an entry to the disk file INVNDX.NSP. The core search takes less than 0.001 seconds CPU time.

The procedure then accesses the disk, looking at every 32nd term entry in the file INVNDX.NSP (on the DECsystem-10 the disk is directly accessible by blocks, of 128 computer words, or 32 four-word records) until it arrives at the correct block. Terms are read in until a match is found, or the term passed. When the search term matches the first 12 characters of a record in INVNDX.NSP the associated record in the continuation file INVPST.NSP is also read.

If a tag is present in the search term restricting the term to a single data type, then the tag of the term in the index must also be checked. A search term may match many entries in the index, and the



Core Data Structure, Inverted Search

Figure 7

postings for each of these index entries must be ORed together. If the entry has fewer than 2,000 postings then the postings are in a postings list of 18-bit numbers. The bit in a temporary bit vector for the search corresponding to each of these posting numbers is turned on. If the term has 2,000 or more postings then the postings are already in bit vector form, and the bit vector representing these postings is simply ORed with the temporary search bit vector. The bit vectors used to hold the intermediate values of the search are always held in core and never swapped out as the bit vectors representing completed searches can be. Because the searches are in product of sums form, only two of these temporaries are needed to conduct any search.

As the search scans through file INVNDX.NSP an asterisk is blinked on the screen to indicate progress. When the first match to a search term is found the bell on the terminal is rung.

The inverted file is built in three steps. The <u>first</u> is to extract the terms from the file of index strings generated by the translation program DINTRN. Each of the index terms tagged in these strings is inverted as it appears in the record with the exception of the free index terms. As with the controlled terms these fields may include phrases, such as ACOUSTIC VELOCITY MEASUREMENT. In order to provide as much flexiblility as possible while searching, and to make the results of the searches by different algorithms be as nearly identical as possible, this phrase would be entered in the inverted index as ACOUSTIC VELOCITY MEASUREMENT, VELOCITY MEASUREMENT, and MEASUREMENT.

Blanks are considered word delimiters[2], except when the blank occurs within slashes. INSPEC uses slashes to delimit special characters, so that H_2O would be entered as H/SUB 2/O; this would be treated as a single word by the term separater.

The output from DINTRM, the program which breaks up the index phrases and attaches the INSPEC sequence number to them to identify their source, outputs the terms into four files on the basis of the first letter to make sorting possible with the limited disk facilities available[3].

The second step is to sort these files of index entries, and then in the third step, the program INVBLD (Inverted File Builder) reads in the terms one by one, generates a postings list for duplicate terms and phrases, and writes this information onto disk files INVNDX.NSP and INVPST.NSP.

No limit is placed on the number of postings an index phrase might have. The length of the phrase is limited to 4,095 characters, longer than any of the actual phrases. Limitations on postings and term length occur in many inverted files, but are avoided to make the inverted search as comparable with the linear and superimposed searches as possible.

---

[2]Since these phrases are not sentences, most punctuation is already eliminated.

[3]Ø-C, D-J, K-Q, and R-Z split the file nearly evenly.

Superimposed Coding from Microstructure--MICSER

The superimposed search is called MICSER for Microstructure Search, because the superimpostion is done on the basis of character structure of the words or what is often called the microstructure of words (see Lynch [50]). This is by far the most novel of the searches and a great many decisions on its basic structure had to be made. (See page 58 for the analysis and development of it.)

After extensive simulations the decision was made to employ a 15-word (540-bit) key; seven words of the key based on bigrams (combinations of two letters), and eight based on trigrams (combinations of three letters). Bigrams are easier to handle than trigrams because of their limited number. If an alphabet of 37 characters is used (26 letters, 10 numbers and a delimiter), this comes to only 1,369 possible bigrams, but 50,653 trigrams. Actually, in these programs it was more convenient to use the standard ASCII codes as much as possible, resulting in a greater number of bigrams to be accommodated. From the ASCII code for '/' to the ASCII code for 'Z' there are 43 different characters and, although only 37 of them are used, this requires tables able to map 1,849 bigrams, and 79,507 trigrams into specific bits in the keys. This is done for the bigrams with a table indicating for each bigram the location of the bit into which it is to be mapped. The trigrams are hashed into a table slightly larger than the number of bits assigned to the trigrams, which in turn specified what bit in the key is to be turned on if a trigram hashed into that value.

A perfect hashing function is one which will map a set of keys perfectly evenly into the table they are to be entered or retrieved from. This program performed its hashing on the ASCII codes of the characters of each trigram to assign the trigrams into specific bits in the key. The hashing function used in the program was arrived at experimentally after trying a number of possiblities, looking for a method which is computationally fast and gives an even distribution of the trigrams into the key. The values of the characters are shifted within a computer word, exclusively ORed together, and then transformed into a number within the range of the number of bits in the table they are to be hashed into. In SAIL this is written:

((C1 LSH 10) XOR (C2 LSH 5) XOR C3) MOD HSZ

where C1, C2, and C3 are the members of a trigram and HSZ is the size of the table.

These mappings are made on the basis of the bigram and trigram counts from 1,000 INSPEC records. To map the bigrams into the key they are sorted by frequency into descending order, and then read in one by one and assigned to the bit in the key which had the least chance of being set. In this way the most frequent bigrams had a bit assigned to them alone, while the less frequently occurring ones are collected together to increase the chances of their bits being set. The trigram mapping proceeded in much the same way, although 'collisions' of high frequency trigrams could occur.

These mappings are stored on disk as arrays of binary numbers and

are read in by both the key file building program and the search program. The file building program reads in each record to be indexed, generates a key by these bigram and trigram mapping functions for each bigram and trigram in the record, and writes the key out along with a 16th word which contains a pointer to the indexed record. The result for a 100,000 record file is 100,000 16-word records. This occupies approximately 25 percent of the space of the original file of index strings.

It is conceptually very simple to match a single term against such a file. A key is generated for the search term in exactly the same manner that the index keys to the file are created. Each index key is then read in and compared to the search key. This comparison proceeds very quickly because every bit postion set in the search key must also be set in the index key in order for them to match. As soon as a bit is found in the index key that should be turned on, but is not, the record corresponding to that key can not contain the search term and the program goes on to the next key. Although it might seem that the longer keys would take longer to scan than short keys, the increase is small because the key seldom has to be searched in its entirety. On the DECsystem-10 36 bits can be checked at once, and slightly more than one-half of the searches fail within the first two words of the key.

If the index key does match the search key in the important bits, its record is read in and matched against the search term. The program does this second match using the same finite state machine technique

used by the linear search, as described in the next section, which scans the record character by character to verify that the record satisfies the search question. A match at this second scan is a 'hit' and a mismatch is a 'false drop.' Often a false drop will contain the needed bigrams and trigrams, but not the term that is wanted. When a match is found this is noted by setting a bit corresponding to that record in the search's bit vector.

This simple algorithm works best if there are only one or two terms to be matched at once against the index keys. With a complex search question with many search terms, the key for each search term has to be matched against the index keys, and their Boolean logic evaluated. To overcome this, a more sophisticated method was developed to match multiple terms against the index. For a search in the form:

[A+B]*[C+D]

the expression is converted to an equivalent sum of products form:

A*C + A*D + B*C + B*D

Each of the products in this formula can be represented by the logical OR of their separate keys. In the example, a key for the term A ORed with the key for C will result in a key which will match only another key which satisfies both the A key and the C key.

Example using a 12 bit key:

A key: 001 010 000 000

B key: 010 010 001 000

C key: 010 000 000 100

D key: 100 000 000 110

A*C key: 011 010 000 100

A*D key: 101 010 000 110

B*C key: 010 010 001 100

B*D key: 110 010 001 110

These resulting composite keys can be broken into groups of three bits. Of the eight possibilities for combinations of three bits:

0 000

1 001

2 010

3 011

4 100

5 101

6 110

7 111

only combinations 2, 3, 5, 6 and 7 will satisfy the search request on the first three bits. For example, combination 0 (000) will not satisfy the search since all four composite keys require at least one bit set in the first three-bit section. For the next three bits, combinations 2, 3, 6 and 7 will satisfy it and no others. Of the 64 possible combinations of six bits only 20 satisfy the first six bits of the example. By building a table with entries for each of the eight combinations for every set of three bits in the key a very complex strategy can be rapidly checked against the index keys.

In the actual implementation, groups of six bits with 64 possible

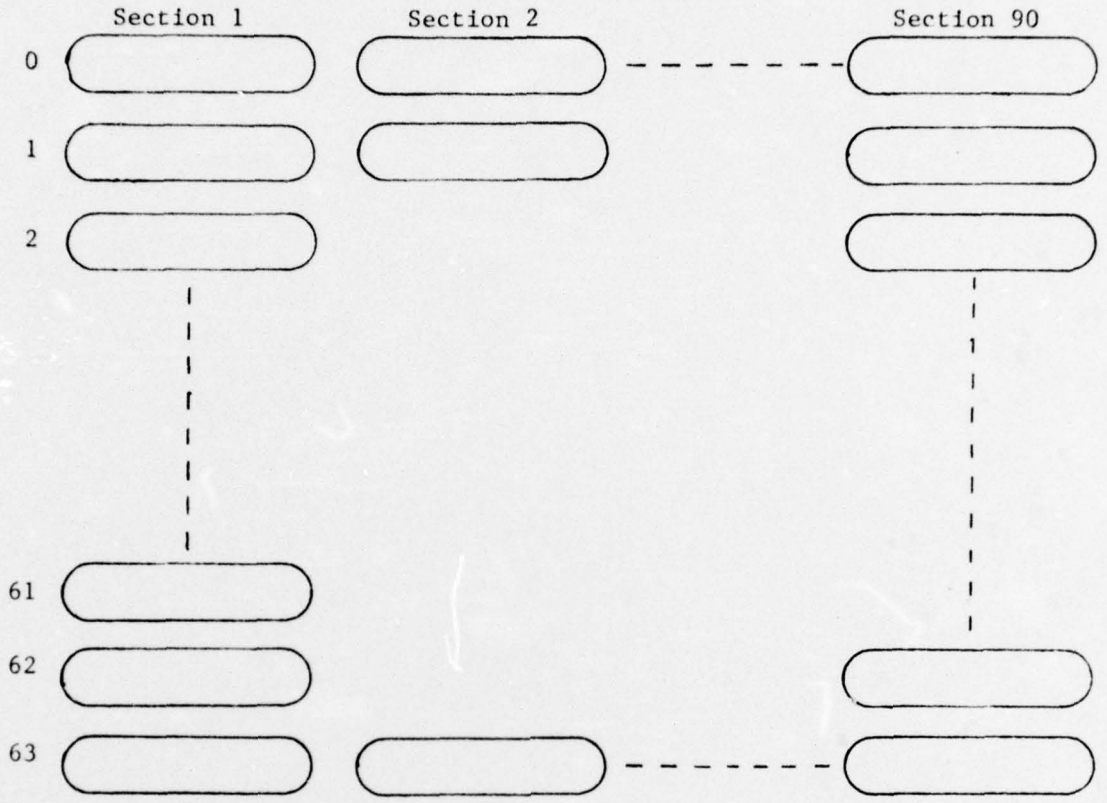
combinations are used, and each entry in the table is a 36-bit word, with bits set indicating which of the composite keys are satisfied by that combination of six bits in that position in the keys (see Figure 8). The entries are ANDed together to keep track of which composite keys are satisfied by all six-bit sections checked so far until (normally) a point is reached where none are satisfied and the program skips to the next key, or until the index keys satisfies one or more composite keys, in which case the actual record is brought in and searched as in the linear search. For an explanation of how this secondary scan is done see the next section on the linear search.

It is conceivable in a very complicated search that more than 36 composite keys could be generated. For example:

[A+B+C+D] * [E+F+G+H] * [I+J+K]

would be transformed into an expression with $4 \cdot 4 \cdot 3 = 48$ terms. In this case only the first two synonym groups (16 composite keys) would be used to generate the table for the key search, representing a match on the keys for the first eight terms. If a potential hit is found on these, then a slower algorithm which is not limited in its number of combinations evaluates the full expression against the search keys to see whether the index key is a match or not.

Because this search sometimes has a slow response time, and often takes quite a while to finish, a series of commands is provided to let the user follow the search as it progresses, and terminate the search to reformulate the search question if the results are not satisfactory.



Each section corresponds to one 6-bit section of the 540-bit key.

Within each section the 64 entries correspond to the 64 possibilities of combinations of 6 bits.

Each entry in this array is one 36-bit computer word, of which as many bits as search terms in the question are used. Each of these bits indicates whether a section in the index key satisfies the search term or not.

Core Data Structure, Superimposed Coding

Figure 8

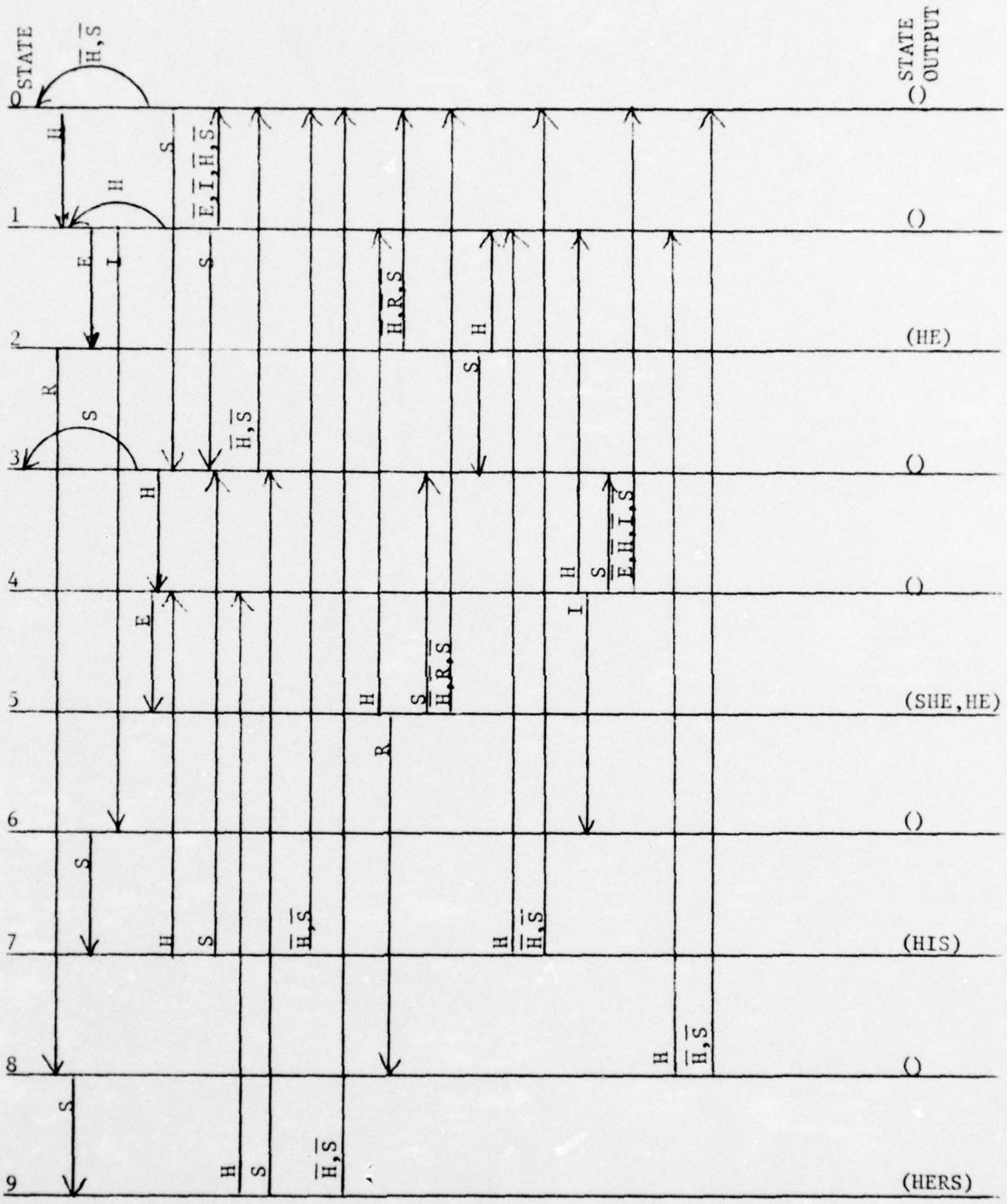
These commands let the user control whether hits are displayed on the screen or not, whether false drops are displayed, and to have a short status report displayed showing the number of records checked so far, the number of hits, and the number of false drops. See Appendix B for a fuller explanation of the use of these commands.

Character by Character Linear Search--LINSER

The linear search simply searches the file of index strings (written by program DINTRN) to which the other searches construct indexes. It uses a fairly new algorithm developed and analyzed by Aho [3]. For the set of terms in a search question a finite state machine is constructed which 'accepts' these terms.

The following is an example taken from [3]. The terms being searched for are 'HE', 'SHE', 'HIS', and 'HERS'. These terms are considered truncated left and right, that is 'HE' would be found in the string 'SHEEP'. Figure 9 is a diagram of the finite state machine needed to match on these four terms. As can be seen, even a very simple search may result in a fairly complicated structure. The complication is in the structure, however, and not in the program's execution. Each letter input simply results in one state transition. Occasionally a state is encountered which matches one or more search terms, and this term or terms is the state's output.

Using the string 'USHERS' for input to the finite state machine, the following state transitions are made:



Matches HE, SHE, HIS, HERS

Finite State Machine

Figure 3

Input: U S H E R S

State: 0 0 3 4 5 8 9

The machine is always started in state 0 at the start of each input string. The transition into state 5 causes HE and SHE to be output as matches, and the transition into state 9 outputs HERS, the third term matched in this string.

If truncation is not wanted, leading and trailing delimiters are added to the search terms. The finite state machine is then set up on the basis of these terms to require the delimiters for matches.

In the implementation of this alogorithm each state in the machine is represented by a 43-element array (each element of which indicates the transition to be made if the character corresponding to that element is input at that point) and an array with a 36-bit word, each bit of which indicates whether a corresponding search term is satisfied by reaching that state (these are the 'outputs' of the states). As the search progresses these outputs are ORed together at every state transition, and after a full record has been input to the machine, this shows which of the terms have been matched. If there are no matches the search proceeds to the next record, otherwise the Boolean logic of the search question must be checked. To make this take as little time as possible, a set of bit masks is constructed before the search begins, one for each of the synonym groups, indicating which terms are in which group. It is then very fast to take the mask corresponding with the first synonym group, AND it with the word representing the output of the

search and see whether any of those terms are present. This continues until a synonym group is found which does not have any members present, in which case the evaluation ceases and the next record is read in, or until all the groups are checked and a hit is found. At that point, depending on switches which can be set dynamically by the user, the citation may be displayed on the screen, as in the superimposed search, and is entered as a hit in the search's bit vector by setting the appropriate bit for that citation.

CHAPTER IV

FILE STRUCTURE AND PROGRAM ANALYSIS

Modules in Common

The routines which all the searches use have little effect on the outcome of this study since, except for the bit vector logic routines, all of the searches depend equally on them. An increase in the time taken to translate the records raises the cost of all the searches equally.

The bit vector logic procedures are contained in module BVLOG. Nearly all these procedures' inner loops have been recoded into assembly language. This was fairly easy to do since the procedures are very simple, and resulted in substantial decreases in run times for these procedures. The routine to AND two bit vectors 100,000 bits long ran in 0.075 seconds when written entirely in SAIL. The assembly language version used by the search programs reduced this to 0.025 seconds, a reduction of two-thirds. The final version had an inner loop of four machine instructions which are executed for each word in the bit vectors. Timings on these individual instructions give a total of 8.4 microseconds for each iteration. The time to AND two bit vectors together is then given by 0.0000084·(Number of bits)/36 seconds. For the bit vectors used in searching this gives 0.024 seconds total, or very close to the measured 0.025 seconds.

Since the inverted search would normally call this routine less

than four times for each search it should account for less than 0.1 sec of the total execution time, but would increase to nearly 10 seconds for a search of 10,000,000 items. Coding this routine in assemble language reduced the run time of the inverted search of 100,000 records approximately 20 percent. On a larger file this routine would take a larger proportion of the total run time of the search, and its efficiency would become more important.

Analysis of this precision is possible only on a procedure as restricted and simple as this. More complicated procedures such as those which search through the inverted index on disk are very file and query dependant and much harder to measure precisely.

Inverted Search

Building the inverted file proceeds in three steps: term separation, term sort, and the actual contruction of the file. It is important to show that the times taken by these programs are not unreasonable in comparison with the time taken to generate the superimposed file, since this is basic to the hypothesis.

In preliminary tests, DINTRM (Directory INSPEC Term Finder), the program that separated the terms for sorting, ran in about 0.04 seconds CPU time per record, and INVBLD which read in the sorted terms and constructed the files took slightly longer, about 0.05 seconds per record. These times compare to an time of 0.03 seconds per record CPU time to generate the superimposed keys. The extra time involved for

each of the steps in the inverted file construction is expected because they involve more character comparisons than the superimposed key program, and are working on files the same size, or larger.

For each record, the term separation program must find each index phrase and write it out along with its record number. For the free index terms this is complicated by the need to separate the phrases into words. Although index phrases are tagged by the translation program to make this as fast as possible, there is still a large amount of character comparison and movement. In comparison, the superimposed key generation program does one scan over the whole index string, ignoring the value of all of the characters except to map them into the key via tables and hashing. The index string is not moved at all, and the only information written out is a 16-word binary array.

Program INVBLD also is forced to do a great amount of character manipulation to contruct the inverted file. Each term has to be read in, compared with the term currently being entered in the inverted file, and either its posting placed in the file, or a new entry started. To save CPU time in this step, the further comparison necessary to eliminate duplicate postings is not done. These duplicates are discarded when read in during the searches. This program does less I/O than DINTRM, but the greater number of string comparisons makes it run slightly slower.

All of these file generation programs are written completely in SAIL with no assembly language blocks. The inverted file programs are

mainly involved in string scanning, comparison and movement, for which SAIL compiles excellent code; little improvement could be expected by trying to code these fairly complicated programs in assembly language. The superimposed key generation program is simpler and is mainly array indexing, something more efficiently done in assembly language than through SAIL. A program with an assembly language version of the key generation procedure was written and found to be more than 30 percent faster than the program written in SAIL. This program was not used since it was felt that a more accurate measurement of the relative times taken by the programs would result if all were written in the same manner.

The sort program used is a new system sort supplied by Digital Equipment Corporation. Before this new sort program was released the system sort available was so slow and wasted so much disk space that the author wrote a sort program for variable length records. The new system sort generates machine code for the comparisons needed, and runs substantially faster than both the old system sort and the sort the author wrote. Although it uses more disk space than some sorts do, this sort appears to be very efficient in terms of run time, and there is no reason to believe that major improvements could be made.

The inverted search program itself runs very fast with most searches running in under one second CPU time. This is fast enough that any improvement in speed would not affect the results of this study, although some analysis of its performance is necessary to obtain an idea of how well it would perform with larger files.

Approximately 20 percent of the total CPU time, or 0.2 seconds, for an inverted search (normally several terms) is involved in reading in the search question, verifying the logic, and parsing it; these are setup times independent of the size of the data base. As each term is passed to the index searcher, an additional setup time of 0.015 seconds is taken to convert the term into a sixbit code, generate masks for right truncation, and determine whether the term is restricted to a certain field.

Locating the disk block at which to start searching the inverted file for the term takes nearly 0.05 seconds CPU time per term, 25 percent of the total search time. Only a small fraction of this time is spent searching the core indexes. The time taken to locate a term's disk block can be expected to rise with the logarithm of the index size, but the vocabulary in this study has already reached a fairly stable size, and doubling the vocabulary size would only increase search times by about five percent.

The main search loop which reads in entries from the inverted file, compares them with the search term, and keeps track of the postings of the records that contain the search term, takes 40 percent of the total time of the search, 0.07 seconds per term. Accurate measurement of exactly how this time is spent is difficult because the loop involves a series of very fast steps repeated for each entry in the inverted file which is compared with the search term. Approximately 25 percent of the time in this loop is spent reading in the postings, and

entering them into the search's bit vector. This 25 percent is about 0.02 seconds run time and is fairly constant if the average number of postings per entry is less than 100. In this experimental file the average was less than four postings per entry, so it would take a very large file before this time would show an appreciable increase. Ten percent of the time in the loop is spent positioning the disk for random access, and the rest of the time is spent reading in the index entries and comparing them with the search term. For files with larger vocabularies the main search loop would take proportionately longer, however such growth in vocabulary can be controlled better than was done in this experimental system; and in any case the vocabulary would not grow linearly with the size of the file.

Superimposed Search

The program for the superimposed search was developed over a period of a year and a half. The first superimposed key used was very simple, using one 36-bit word per title from a MARC tape. Experiments were also made with a half word (18-bit) key in which the most common letters were discarded, and the least common collected, giving a more even distribution of bits, this being intuitively what is wanted.

It is clear that any bits in the key which are always the same value in all the keys give no new information about a record. For example a bit which is set if the letter E appears in the title of a book is of little use in a search, since the great majority of titles

will have one. It offers no reason to reject a title from further scanning. Much the same argument applies to Z. Being able to tell from the key that a title does not contain a Z is very useful if the search term contains one, but unfortunately few do, so a bit devoted to indicating the presence or absence of Z will normally go unused. Another example of a bit of little use, for title keys, is the U bit when the Q bit is set, since in this case it will nearly always be on, and provides no additional information. This correlation between the Q and U bits reduces their value in searching.

The most useful bits in a search key are those which are set in half of the records, and are independent of the other bits. The information content of a key can be measured in terms of its relative entropy. Shannon [66] first developed the mathematics of entropy as a measure of information in communication theory, and others such as Lynch [49] and Fokker [28] have applied this to search keys, and Mooers [55] has applied it specifically to superimposed coding of subject codes. Entropy is measured by the formula:

$$H = -\sum p_i \log(p_i)$$

Where p_i is the probability of each possible code occurring. Entropy is at a maximum when all the probabilities are equal, in which case it is equal to the log(n) where n is the possible number of states and the logarithm is normally taken to the base two.

If a message is encoded in the 256 different eight-bit characters, each of which has an equal probability of occurring, then the code's

entropy is log(256) = 8. To compare codings of different length, it is often more useful to normalize entropy to relative entropy. Relative entropy is the actual entropy measured, divided by the maximum possible entropy for that coding, a number ranging from 0, denoting no information content, to 1.0 indicating the maximum possible.

For an illustration of this with codes two bits long, suppose each of the codes occurs an equal proportion of the time. The coding then has an entropy of 2.0 and a relative entropy of 1.0:

Code	p_i	$\log(p_i)$	$-p_i\log(p_i)$
00	0.25	-2.0	0.5
01	0.25	-2.0	0.5
10	0.25	-2.0	0.5
11	0.25	-2.0	0.5
			2.0 Total
			1.0 Relative entropy

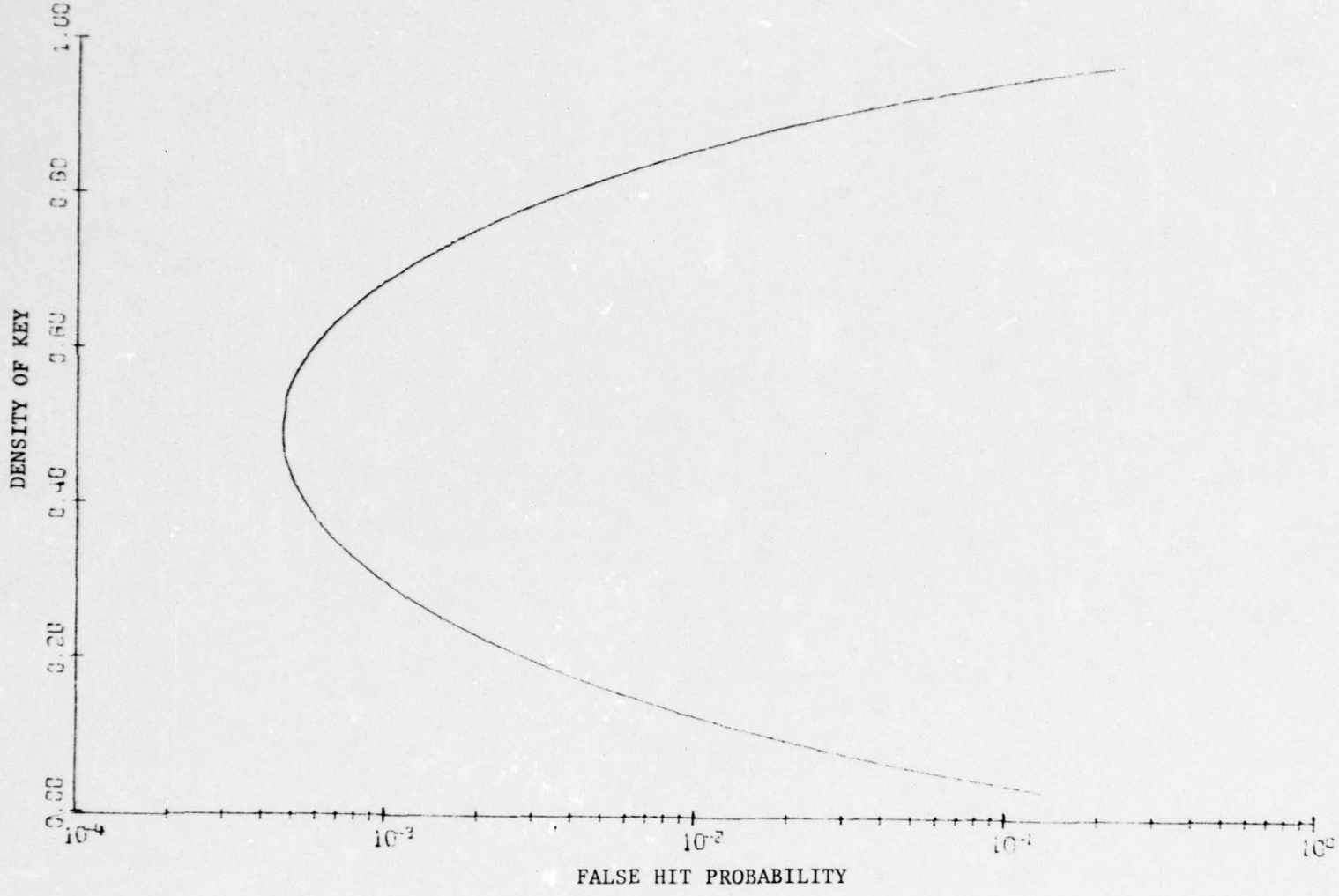
However, if the frequencies are not equal:

Code	p_i	$\log(p_i)$	$-p_i\log(p_i)$
00	0.20	-2.32	0.464
01	0.20	-2.32	0.464
10	0.30	-1.74	0.521
11	0.30	-1.74	0.521
			1.971 Total
			0.985 Relative entropy

The inequality of the frequencies lowers the entropy of the code.

In the superimposed search each 540 bit key represents a code. As one can see by inspecting the two-bit codes, if each possibility occurs, on the average, the same number of times, an equal number of zeros and ones will be set. In the keys this corresponds to having half the bits, on the average, turned on in the keys. The effect of having the key half 'full,' a density of one-half, can be graphed for a very simple case. Suppose the key is 540 bits long and the index strings are an average of 100 characters long. By varying the number of bits set in the keys for each bigram, the density of the index key can be varied. (See Figure 10 for a graph of this.) As can be seen, the minimum false drop probability occurs when the key's density is one-half. At this point the key contains the maximum amount of information possible. Lynch [49] has measured the entropy of keys similar to these by calculating the entropy solely on the frequency of the individual bits. This technique ignores correlations between bits which reduce the entropy of keys by reducing the chances for certain configurations to occur. A more realistic measurement of the entropy of keys can be obtained by measuring the entropy of randomly selected pairs of bits over a number of keys and taking their average.

A relative entropy of 0.85 is achieved with the actual keys used in this study. This compares with an entropy of 0.80 for the keys used in preliminary studies on a simpler file with less effort made to obtain a high entropy. The average density of the keys used is 0.45. I expect that it may be possible to achieve a relative entropy of more than 0.90 using superimposed keys based on bigrams and trigrams.



Effect of Key Density on False Drop Probability

Figure 10

A relative entropy of 1.0 will result in the lowest false hit probability for a key. Assuming random strings and a random mapping of them into the keys, Harrison [32] has derived the following equation for the false drop probability of the key:

$$FHP(L1,L2,M) = (1-e^{(-L2/M)})(M-M\cdot e^{-L1/M})$$

Where L1 = Number of bits set by search term

L2 = Number of bits set by index string

M = Length of the key in bits

e = Base of the natural logarithms (2.718...)

This formula is derived from the fact that the number of zeros in the key length M bits of a string which sets L bits will be $M\cdot e^{-L/M}$, so that $M-M\cdot e^{-L1/M}$ is the number of one bits present in the search key, and $1-e^{-L2/M}$ is the number of bits set in the key of the index string divided by the number of bits in the key to get the index key's density. For small L1, $M-M\cdot e^{-L1/M}$ is about L1, since the chance of a short search term setting a bit twice in a long key is small[1]. In this case the false drop probability is simply D^{L1} where D is the density of the key. The key is very sensitive to key length if the entropy can be kept constant. If the length of a key is doubled, and a density of one-half is to be kept, the number of bits set in the key per character of the index string must also increase. If the false drop probability of a key, of density one-half with a search term which sets eight bits, is

---

[1]For example in a 540-bit key a search term that is mapped into 16 bits in the key will set 15.77 bits, on the average.

approximately 0.004, and if the number of bits set by that search term in its search key is doubled to 16 bits by lengthening the key, the false drop probability decreases to 0.000016, effectively squaring the screening power of the key.

Lengthening the key is done at the expense of extra storage space, slightly slower key scan execution times and greater complexity in the key. The greater complexity is needed to generate more bits per character, and to maintain a maximum entropy. Because of the nature of n-grams this is difficult to achieve, and in fact, other things being equal, a longer key is bound to have a lower relative entropy than a short key in which there is better control over the bit frequencies.

To investigate the effect on run time that the lower false drop probability of longer keys would have, simulations of keys of various lengths were done. Keys up to 1116 bits (31 36-bit words) were found to be feasible in simulation studies, and are most useful with short search terms which set few bits in their search keys, and therefore have high numbers of false drops.

To simulate the efficiency of different superimposed keys, bigrams and trigrams are counted from a sample of 1,000 INSPEC index strings. In each string it is only the first occurrence of a given bigram or trigram which is important since subsequent occurrences simply reset the same bit. Using first occurrence counts, an accurate false drop probability can be calculated for a key for a given number of bits. In the bigram simulator, a key is built up with a near optimum mapping, in

terms of the bigram frequencies, into the key. The proportion of time each bit is expected to be turned on is measured, and the false drop probability for a search term of length L1 is calculated by:

$$FDPRB = \prod F_i^{L1 \cdot F_i/TNB}$$

Where F_i = Frequency of bit i

TNB = Total number of bits set in the key

$= \sum F_i$

$L1 \cdot F_i/TNB$ = the chance that bit i will be set.

The run time for a superimposed search is the input time for the file of keys, plus the key scan time, plus the time needed to input the index strings indicated as possiblities by their associated keys, plus the time needed to scan these strings to determine whether they are hits or false drops. In simulating the search, since the number of false drops is expected to be quite a bit larger than the number of hits, and since the number of hits to be expected is not known, only the time taken to check false drops is simulated.

The CPU time needed to read in the keys can be measured directly, and this was done, timing it against the blocking used on the keys. There are decreases in input time as the block size is increased until the block size becomes greater than 10 K words. A blocksize of 16 K words (1024 15-word keys, or 128 disk blocks) was used in the simulations and actual program. This gives an input time of about 1.2 seconds CPU time, and a minimum daytime of about 51 seconds[2].

---

[2]In the actual searches four out of 49 ran in under 51 seconds clock time.

The time needed to scan the keys can be estimated on the basis of key length and false drop probability, for these control how many sections of the key need to be looked at. The key scan section of the superimposed search is written in assembler and the instructions were timed. For maximum efficiency each section of the first word of the key is coded separately, not in a loop. The time needed to scan the first word of the key is:

$$16.8 + \sum 7.6 \cdot PRS \text{ microseconds}$$

where PRS is the probability of reaching that section, which is just the probability of a false drop on the preceeding bits and the is over the number of sections in the key. The false drop probability per bit is: $FDPRB^{1/NBINK}$ where FDPRB is the false drop probability for the whole key, and NBINK is the number of bits in the key. The false drop probability for a section of the key is $FHPRB^{NBINS/NBINK}$, NBINS the number of bits in the section.

After the first 36 bit word of the key, the key scan time is

$$\sum KSTPW \cdot PRW \quad \text{(Sum w=2 to no. of words in key)}$$

where KSTPW = the key scan time per word which

= 6.7+12.6·NSPW microseconds

where NSPW = the number of sections per word

= 36/number of bits in each section

and PRW = probability of checking that word

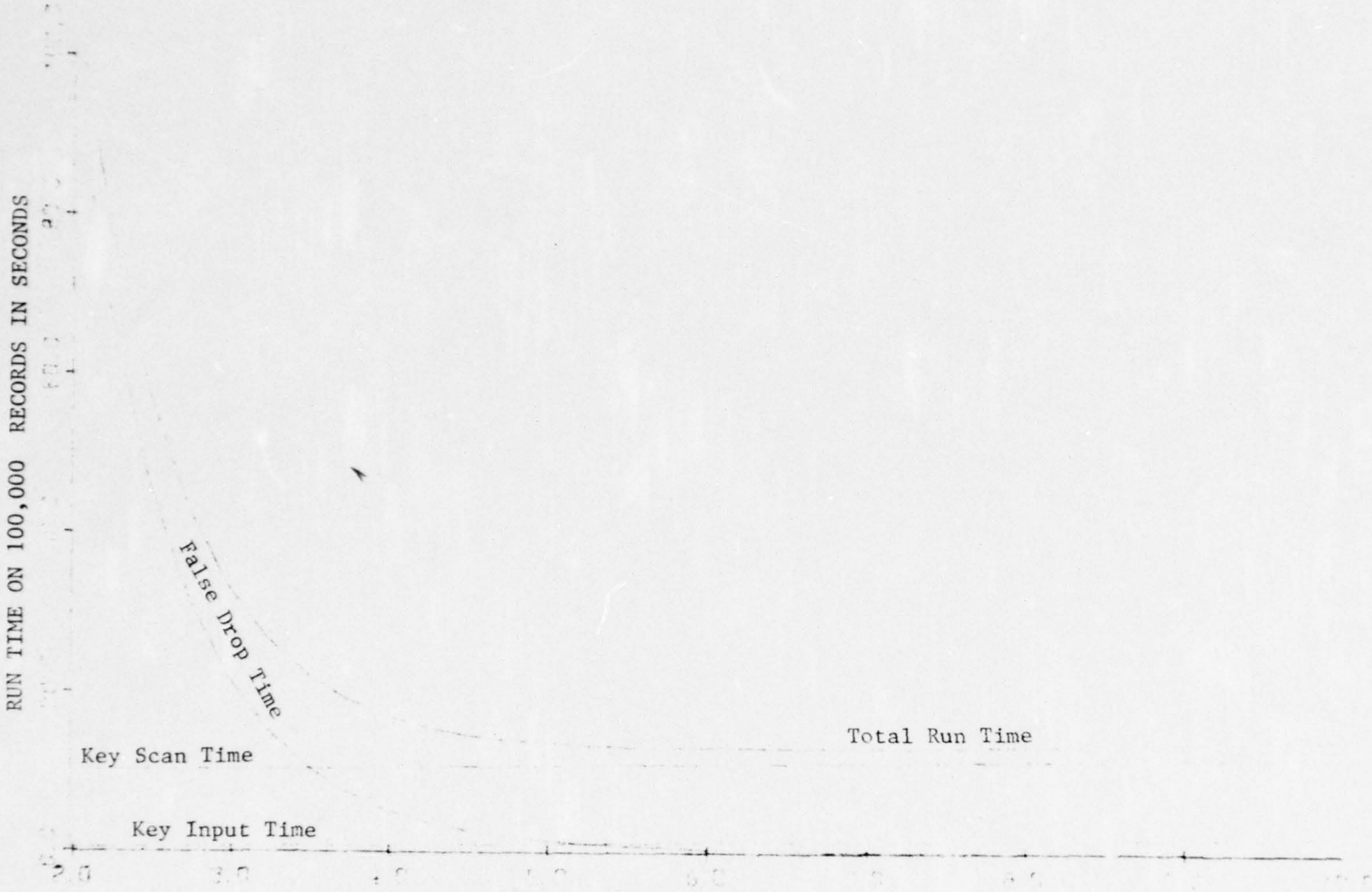
= $FHPRB^{w \cdot 36/NBINK}$

Timings for inputing and scanning the false drops' index strings

were based on a preliminary study in which a program was written to do a superimposed search on 60,000 Science Citation Index titles.

If the length of the expected search term is known, the key length with the minimum run time can be found by simulating the search for various key lengths. (For an example of this see Figure 11.) This graph is based on the assumption that a key of perfect entropy can be generated for any length needed; a search term of the same length as the search term 'ØCOMPUTERSØ' is used. As can be seen from the graph, the minimum total run time occurs with a key of about 650 bits. Beyond this length, the longer time needed to read in the keys and scan them takes more time than any increase in screening power can offset.

The calculations for this graph are based on timings of the search program for scanning the keys and false drops, with the false drop probability simply 0.5^B, B being the number of bits set in the search key, each of which should eliminate one-half of the file from further consideration. The number of bits set in the search key is an estimation based on the number of bits per character which the index strings would have to set in the index keys to obtain a key of density one-half. A key M bits long needs an average of $-\ln(0.5) \cdot M$ bits set in it to reach this density. Several index strings were studied and it was found that approximately 20 percent of the characters in a given index string occur in repeated words and do not contribute to the bit density of the index key because of this redundancy. For example an index string had both RELAXATION TIME and VIBRATION RELAXATION TIME as free



Effect of Key Length on Run Time

Figure 11

index terms. This reduces the effective length to about 247 characters[3]. Another factor entering into the estimation of the number of bits set in the search key are the impossiblity of generating as many bits per character for short strings, such as occur in search questions, as in long strings such as the index strings. This is because a string has one less bigram and two less trigrams than it has characters. In a short string this loss can amount to more than 25 percent of its length. This is adjusted for in the calculations.

No set of keys will have a perfect entropy. The entropy of the keys actually obtained is in the range of 0.8[illegible] to 0.90, and this seems fairly easily obtainable, with this type and length of key. Figure 12 is a graph of the effect on run time that various entropies have. As the entropies decrease the run time rises, especially for short search terms, the increase in run times is substantial. These curves assume that a key of entropy 0.5 has half the screening power that a key with perfect entropy has.

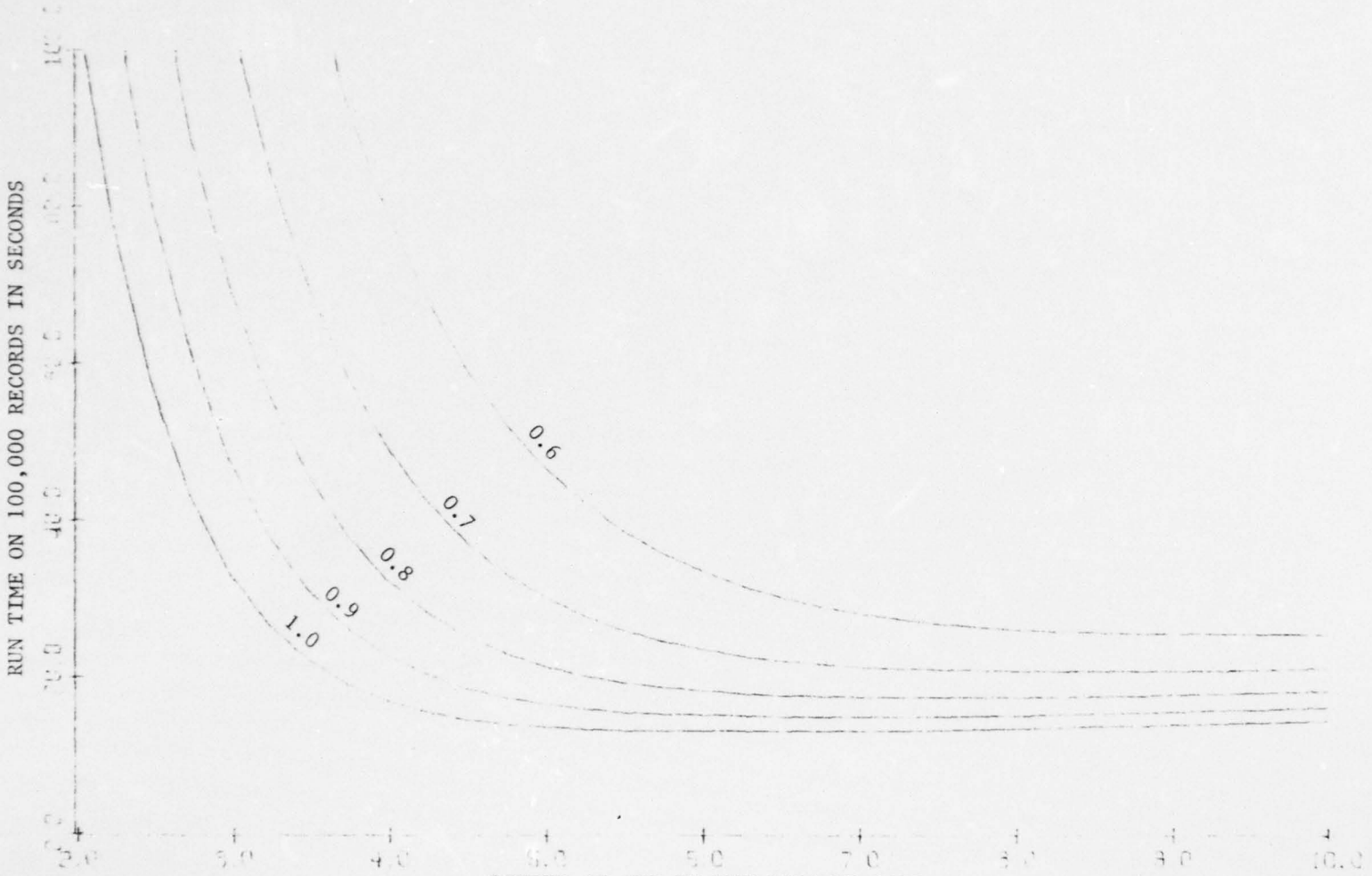
An attempt was made to calculate the expected false drops for more complex searches by summing the false drop probabilities within synonym groups, and multiplying that of different groups. In a search

$$(A + B) * (C + D)$$

if all four terms each had a false drop probability of 0.01, then A+B would be expected to have approximately double that, and the two groups

---

[3]This is consistent with the number of bits actually set in keys based on bigrams and trigrams.



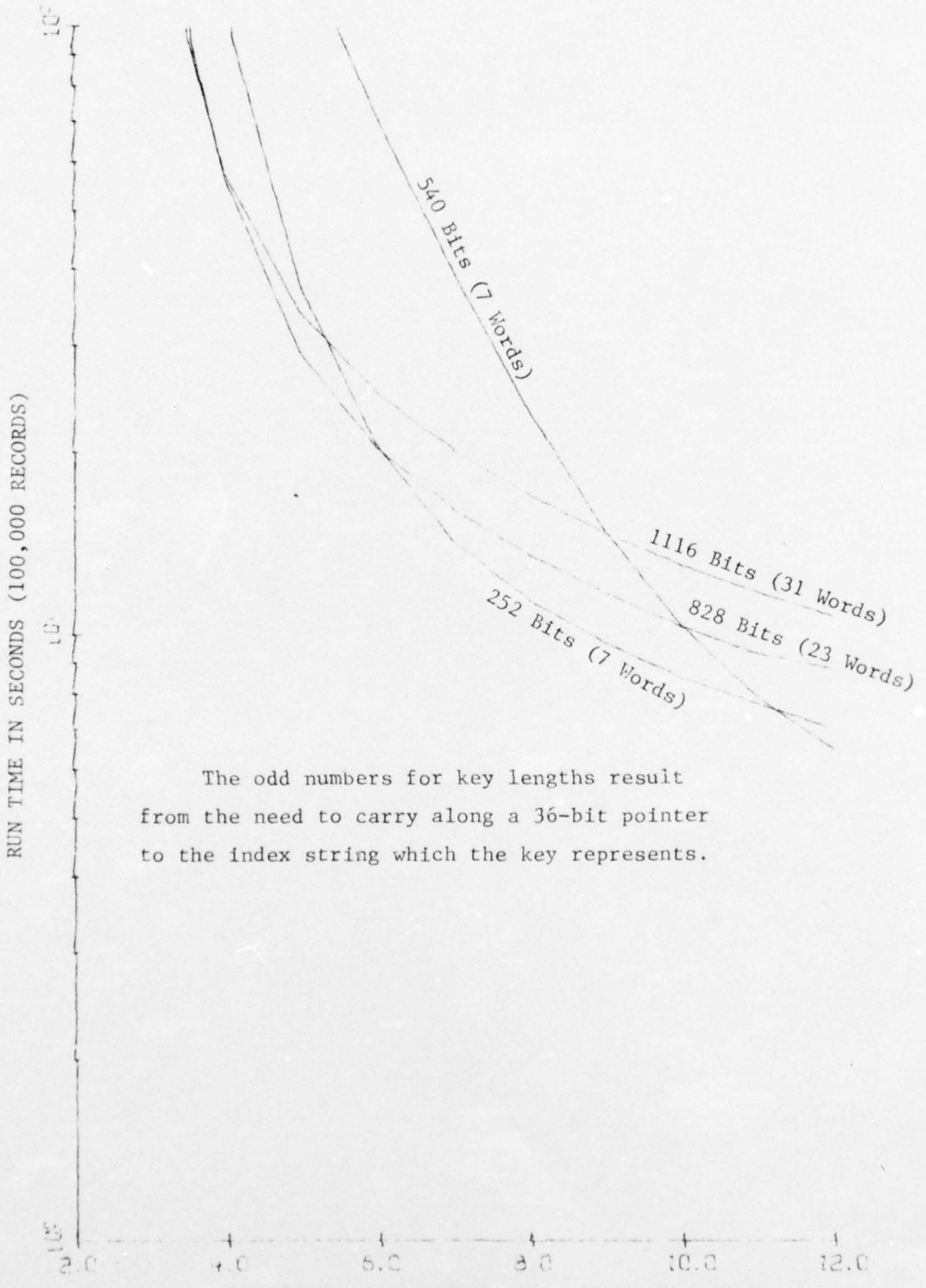
Run Time for Keys of Various Entropies

Figure 12

together, the sqare of that double, or 0.0004. The false drop probabilities calculated in this way are much lower than those actually measured, evidently because this is a case where assumptions of randomness are not valid. Since no other theoretic basis for predicting these probabilities could be discovered, simulation of complex searches has not been done.

The false drop probability, and consequently the search time, for a large number of different keys for searching INSPEC were simulated, all based on actual mappings of bigrams and trigrams. The keys ranged from seven-word bigram keys to 31-word keys with separate fields for bigrams and trigrams. (See Figure 13 for a graph of search times expected for a bigram key of 7, 15, or 23 36-bit words.) In general the shorter keys have an advantage if the search term is very long, setting many bits in its key, while the longer keys do better with search terms which tend to produce a great number of false drops.

There is a strong advantage in using both bigrams and trigrams in a single key in that this allows a string to set more bits in its key, making a search key more selective, and for long keys coming closer to the optimum density of one-half. In evaluating keys which combined bigrams and trigrams into one long key the assumption is made that the bits set by the bigrams would be nearly independent from the bits set by the trigrams, and therefore that the false drop probabilities of two separate bigram and trigram keys can be simply multiplied to get the false drop probability for the combination. This is not entirely true,



The odd numbers for key lengths result from the need to carry along a 36-bit pointer to the index string which the key represents.

Effect of Search Term Length on Superimposed Search Run Time

Figure 13

but entropy studies on the actual keys generated have shown there is no measurable degradation in the entropies of the keys when combinations of bigram and trigrams are used.

All of the key structures which were simulated, and the one which was finally used in this study, kept the bigrams and trigrams in separate sections in the key. This is easier to analyze but it may well be that a key in which both the trigrams and bigrams could appear in any section of the key would be an improvement. One way in which this could be done would be to hash the trigrams into the whole key. Then the bigrams, each of which can be individually mapped, could be assigned to the bits which are expected to have the lowest frequency, giving a very even bit distribution.

The final key selected is a 15-word (540-bit) key, seven words of which are constructed from bigrams, and eight words from trigrams. A 23 or 31-word key would have been slightly faster in some circumstances, but it is judged that the increased storage costs on disk and the increased clock time needed to read the keys in for searching more than offset savings in run time. Search times were expected to range from seven to 50 CPU seconds[4]. A 31-word key would have been expected to have run times range from 11 to 36 seconds[5]. A seven-word key would have performed worse in nearly all the searches.

---

[4]Actual searches ranged from 7.2 to 157 seconds.

[5]The simulation implies that 157 second run time for a 15-word key should run in 70 seconds if a 31-word key is used.

Some economy in core storage could be realized by scanning the keys in four-bit sections (16 possiblities) instead of six-bit sections. The four-bit sections would save more than three thousand words of core, but would run approximately 30 percent slower than the six-bit sections. Alternatively, using nine-bit sections would take 25 thousand more words of core storage than six-bit sections and would run 20 percent faster than the six-bit sections.

The possibility of batching superimposed searches was investigated. With an average run time of nearly 30 seconds, approximately one-half the time is spent scanning the keys and the other 15 seconds of run time reading in the index strings (both false drops and hits) and doing a final scan on them. Running a number of questions in a batch would increase the key scan time somewhat, and increase the final check time in proportion to the number of search questions batched. The maximum scan time for the keys, assuming that all search terms can be checked in one scan, is about 85 seconds for a 540-bit key. The run time per question would then be:

$$(85 + 15 \cdot NSQ + 0.3 \cdot NSQ)/NSQ \text{ seconds}$$

Where NSQ stands for the number of search questions. For 100 search questions this gives 16.2 seconds per search. Although this is a substantial gain over the 30 second average for single searches, it is nowhere near the amount which is saved in the linear search by batching. It would normally not be worth the effort in programming to achieve such small gains in run time.

Linear Search

As mentioned before, the linear search algorithm is based on one devised by Aho [3] for fast pattern matching in strings. This algorithm is based in turn on work by Knuth et al. [42]. The basic algorithm is the fastest way yet found to match a group of search terms against a large file in one pass, but it does not take into account the cost of evaluating the Boolean logic associated with the search. It will be shown below that this time is small compared with the search, in the typical case of a bibliographic search where the chances of a particular citation being a hit are small.

The set up time for the search takes about 0.22 seconds CPU time. This includes reading in the question, checking the question's logic, parsing the question and setting up the finite state machine used in the scan, everything except the search itself. This amount of time is so small compared with the search times for a file of moderate size that no more will be said, except to note that it is linearly related only to the length of the search question, and not dependant on the size of the data base.

The scan loop of the search program is coded in assembly language with an inner loop of eight instructions which must be executed for each character. This loop includes a translation of each character to map all non-alphanumerics into the character code which represents all delimiters. Timings on individual instructions gave a total of 20.5 microseconds CPU time needed per character. For an INSPEC index string of 309 characters this is 0.0063 seconds per record.

Boolean evaluation adds suprisingly little to this figure. Although not written in assembly language, the code produced by the compiler was inspected. On the reasonable assumption that the chances are small of having any one record being a hit, the loop which checks each synonym group in turn will normally only have to check the first such group. This involves about 10 to 12 machine instructions with a minimum of three memory accesses. (Instructions which access memory are normally slower than those which do not.) The Boolean evaluation amounts to only about 25 microseconds execution time, plus a few instructions to pass the value back to the calling routine.

Reading the records into core takes a substantial amount of the CPU time for the total search. Input timings show that it takes from 12.2 to 17.1 microseconds to input one 36-bit word if the records are 300 characters (62 36-bit words) long, about three microseconds per character, or 15 percent of the the time needed to scan the characters. Because the input and computing are overlapped, a device with a faster transfer rate than disk would decrease the clock time of a search about 25 percent at the most.

Based on these timings of the critical (in terms of run time) sections of the program, the estimate of the linear search's run time is:

$$((20.5 + 3) \cdot ARL + 30 + 60) \cdot NR \text{ microseconds}$$

where 20.5 = Time needed to scan each character

3 = Time needed to input each character

ARL = Average record length

NR = Number of records

30 = Boolean logic evaluation time

60 = Procedure call overhead, flag checking, etc.

For the INSPEC data base of 100,000 records this formula predicts a time of 735 seconds, five percent lower than the observed run time of 775 seconds. This may be accounted for by slightly slower input speeds, or very possibly more overhead than is estimated.

Many linear search programs are able to handle multiple questions batched together, so it is interesting to try to extend these timings to this more complex situation. The algorithm must be expanded to handle more terms, and the logic to handle many Boolean equations. The most straightforward way to do this is to have several words which could hold more search terms, instead of a single word which can handle only 36 search terms for the state output. In this case the search logic would execute an extra instruction for each group of 36. Run time for the scan would then be

$$18 + ((NT + 35) \% 36) \cdot 2.6 \text{ microseconds/character}$$

where NT = Number of Terms

and % stands for integer division,

to the largest integer

The Boolean logic check would still be expected to fail the first

time for each question. If the same algorithm was used and simply repeated for each search question it should take about 25 microseconds per search question. With setup time of 0.2 seconds per search question this results in a total run time of:

$$((18 + ((NT + 35) \% 36) \cdot 2.6) \cdot ARL$$
$$+ 25 \cdot NSQ + 60) \cdot NR \text{ microseconds}$$
$$+ 0.2 \cdot NSQ \text{ seconds}$$

where NSQ = Number of search questions

For a search of 100 questions, eight terms in each on the average, on 100,000 INSPEC index strings this would take 2,760 seconds (45 minutes) CPU time, only 28 seconds per question.

A more efficient implementation would simply check whether the state entered has any output at all, since most state transitions will not result in any matches on search terms, and only check the output list if there is a match on at least one term. In this case the scan time would only be slightly more than for a scan for a single search question, and the total time would be

$$(30 \cdot ARL + 25 \cdot NS) \text{ microseconds} + 0.2 \cdot NS \text{ seconds}$$

For the above example of 100 searches this would run in about 1,100 seconds, or 11 seconds per search question.

Unfortunately both these algorithms would probably take an inordinate amount of core storage to run since more than 3,000 states would probably be needed, each state taking 65 words of storage. The storage occupied by the finite state machine could be greatly reduced,

at some expense in run time, by storing the less frequently used state transitions and outputs in a more compact manner.

CHAPTER V

EXPERIMENTAL RESULTS

Data Base and Generated Files

From the INSPEC tape numbers 2 through 20 of 1974, 100,000 records were used as the data base against which the searches were run. These records represent more than nine months of citations for INSPEC. Every readable citation on the tapes was used, although, because of physical read errors, 24 records had to be skipped, 17 of which were on one especially bad tape. The records averaged 26.3 terms per record with an average term length after processing the free index terms as described for the inverted search, of 13.4 characters. These 'terms' can actually be phrases of two or more words, and a great number of them were simply six letter codes for subjects or CODEN's. A total of 2,633,949 terms were inverted for the inverted file. The frequencies of these followed the expected Zipf's [81] curve (see Figure 14).

The 'index strings' averaged 309 characters per record. The inverted file builder found 672,248 unique terms (or types). This is a very large number, and with other methods of choosing the terms for inversion it might be possible to reduce this, although INSPEC is noted for its unusually large vocabulary. Slightly more than 77 percent (518,987) of the terms occurred only once in the file[1]. The most common

---

[1]This could be compared to figures for Toxline. For 374,000 references 687,484 unique terms were found, 55 percent of which occurred only once. Terms were limited to single words with a maximum of 36 characters [23].

Cumulative Counts of Posting Distribution

Figure 14

term was the word 'SYSTEM' which occurred 7,593 times as a free index term, followed by 'MODEL', also a free index term, occurring 6,029 times. In fact, all terms occurring more than 2,000 times (the cut off point to be stored as a bit vector) were from the free index field except two: A9126 a sectional classification code for 'Physical Metallurgy--Treatment of metals, mechanical strength,' and ZGEGAJ the unified classification for the same subject. Both occurred 2,022 times.

Due to the low (2,000 postings) cut off point at which postings are stored in bit vector format, the bit vectors actually used more space[2] than postings in half words would have taken. If the optimum crossover from a space viewpoint had been used (5,558 postings), only three bit vectors would have been used in the entire inverted file with a slight space saving.

File INVNDX.NSP, the section of the inverted file with the first 12 characters of a term plus its length and posting count, occupies 21,000 blocks (13,445,000 characters) of storage on disk, and the continuation file with the rest of the term and postings takes 23,400 blocks (14,960,000 characters) on disk.

The size of the inverted file prevented the possibility of storing the full references[3] on the same disk pack, so a shortened reference[4] was created along with a primary author file (12 characters per fixed length record) for display, demonstration, and output timing purposes.

---

[2] 88,928 words vs. 53,747 words.

[3] 45,000 blocks (280 characters per record).

[4] 24,053 blocks (154 characters per record).

The size of the file which is searched by the linear search, and indexed by the superimposed and inverted searches is 51,172 blocks (328 characters per record) long. The inverted file is 87 percent of the size of this, and the superimposed index is 24 percent. The total record size of the data base including index terms, partially formated references and abstracts is 177,000 blocks (1,133 characters per record). The inverted file increased the storage needed for this by 25 percent, the superimposed search by seven percent, and the linear search by 0 percent, since it did not have any indexes.

Linear Search

Preliminary runs showed that the linear search had to be severly limited in number of questions run in the interest of economy. A total of nine searches was selected, one from each strata of the search questions.

Run times for the nine searches ranged from 771 seconds to 780 seconds, averaging 775 seconds with a standard deviation of 2.3 seconds and an estimated 90 percent confidence interval of between 773 and 776 seconds. The search times for 100,000 records are 0.00775 seconds per record, or 25 microseconds per character. The nature of the search algorithm, which is very insensitive to the question asked, accounts for such a small range of search times, and gives a very good indication of the average search time even though only a small number of searches were performed.

The clock time was also nearly constant, ranging from 925 to 1,173 seconds, with all but one search running in under 950 seconds, and averaged 957. Daytime can be strongly affected on a time sharing system such as the DECsystem-10 by competition for the data channel to the disk, and can easily account for this range of times.

Disk reads were high, 50,518 per search, since the whole file had to be brought into core.

Initial response time, the time at which the first results from the search would be shown, would be approximately 46 seconds, based on the clock time taken by the searches and the averge number of hits found by each search. This makes such a linear search system nearly unusable as an online interactive search system with this size and type of file.

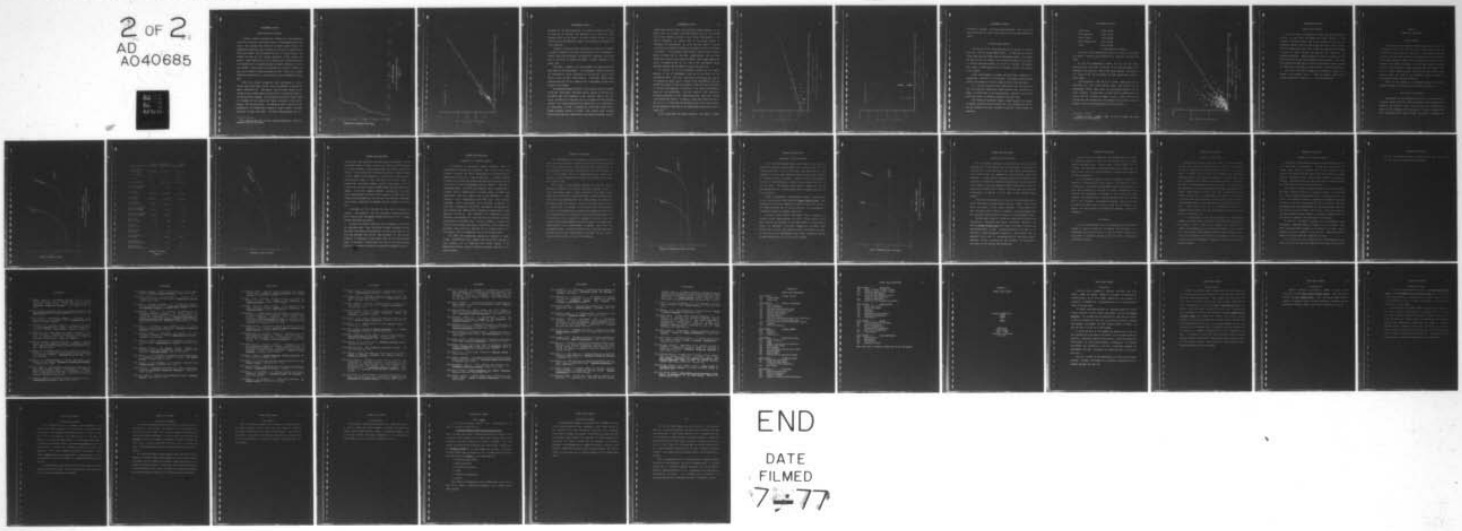
The same set of nine questions was run through the initialization steps used by the linear search, and took an average of 0.22 seconds CPU time. This time is principally in the search question logic verification, question parsing, and finite state machine construction.

The only file building time applicable to the linear search is the time used to translate the tape from BCDIC into ASCII and put it into the format searched by the linear search. This took 19,328 seconds, or 0.193 seconds per record. At least a third of this time was taken up in the collecting of fairly detailed statistics regarding the file.

With a search time of 0.00775 seconds compared with 0.193 seconds per record translation time, 25 searches would need to be done on the file before the searching CPU time would amount to more than the file preparation time.

AD-A040 685 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
SUPERIMPOSED CODING VERSUS SEQUENTIAL AND INVERTED FILES.(U)
MAR 77 T B HICKEY DAAB07-72-C-0259
UNCLASSIFIED R-761 NL

2 OF 2
AD
A040685



END

DATE
FILMED
7-77

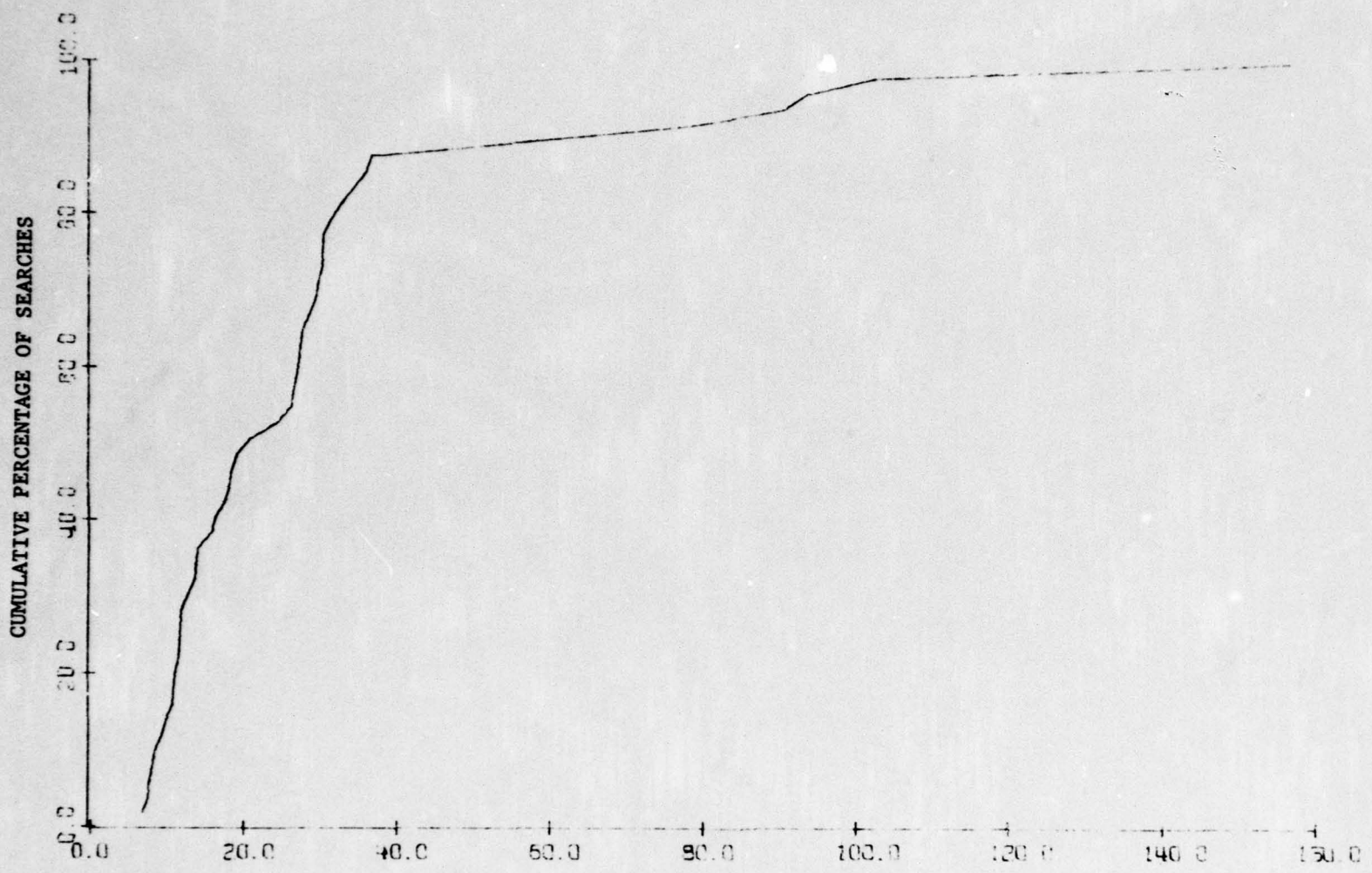
Superimposed Search Results

A greater number of searches were possible with the superimposed coding than with the linear search because the superimposed search ran faster. More searches were needed for accurate timings because the superimposed search has a more variable run time than the linear search. Forty-nine searches were run against the file of 100,000 records. The average run time was 29.7 seconds per search (0.0003 seconds per record). Times ranged from 7.2 to 157 seconds with a standard deviation of 29 seconds and a 90 percent estimated confidence interval of 23.9 to 36.5 seconds[5]. Fifty percent of the seaches ran in under 20 seconds CPU time, and 90 percent in under one minute CPU time. Figure 15 is a graph showing the distribution of the run times for the superimposed searches done.

Clock time closely followed CPU time (correlation of 0.99), running about three times the CPU time, and averaging 95 seconds per search (see Figure 16). This search, like the linear search, but in contrast to the inverted search, does not have to be completed before it can give an initial response. By dividing the clock time a search takes by the number of hits found by the search an estimate of the initial response time can be obtained. The average of these times for the 49 searches was 19.3 seconds with a minimum of under a second. The worst case would be about one minute. Of course these questions are not

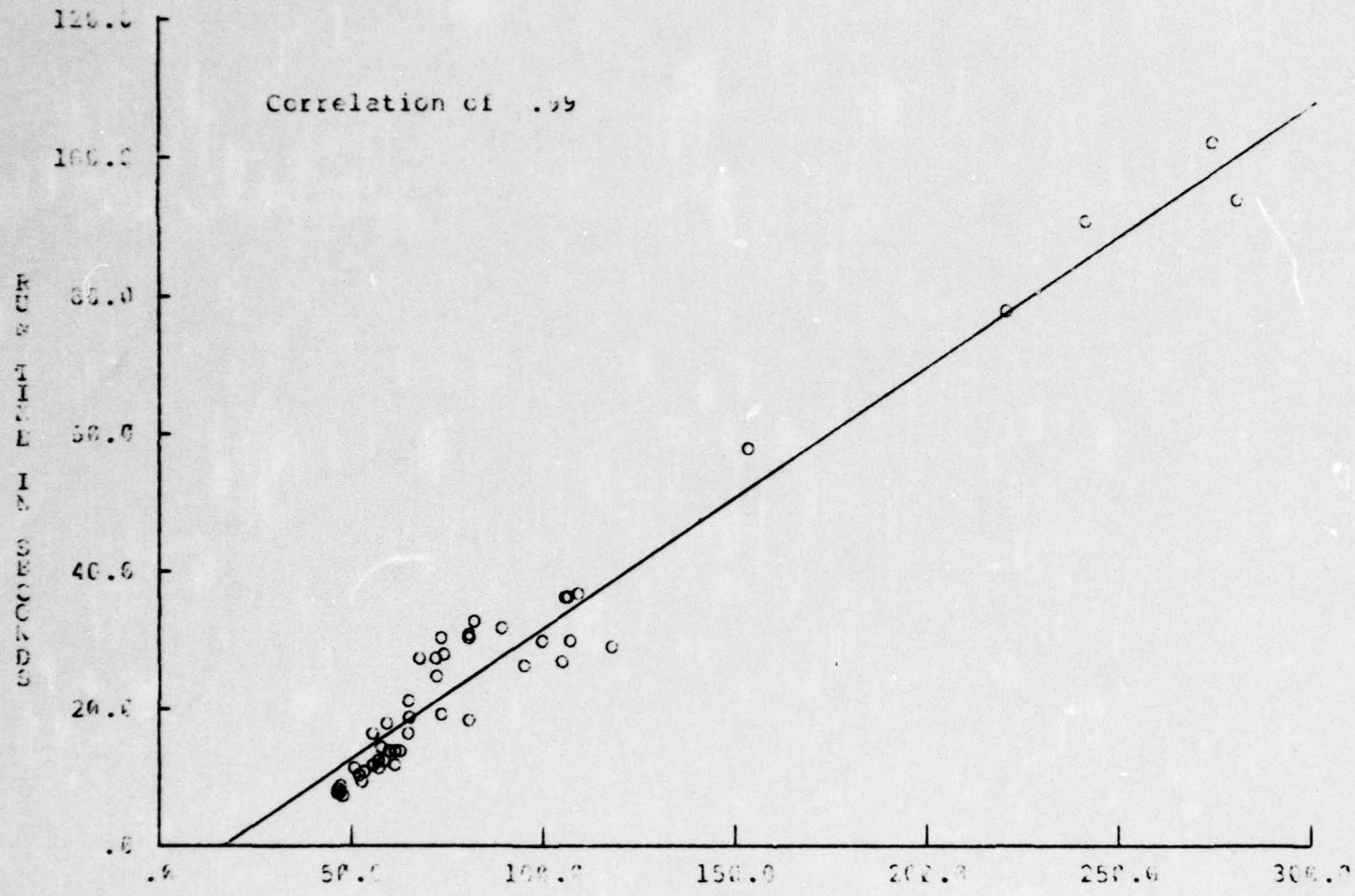
---

[5]The questions were run in three batches averaging 32, 28 and 27 seconds of run time per search.



Run Times for Superimposed Searches

Figure 15



Superimposed Run Time vs. Clock Time

Figure 16

optimized for the search algorithm. An accurate estimate is difficult to make, but by tailoring the questions to the system the worst questions might well be made to run twice as fast as they do now. Experience with the system has shown it to be marginally acceptable for online searching.

Included in these run times are the set up time of 0.22 seconds per search discussed in the linear search results, plus an estimated 0.1 seconds setup time for setting up the keys for searching. This setup time of less than 0.4 seconds per search is small compared to the search times.

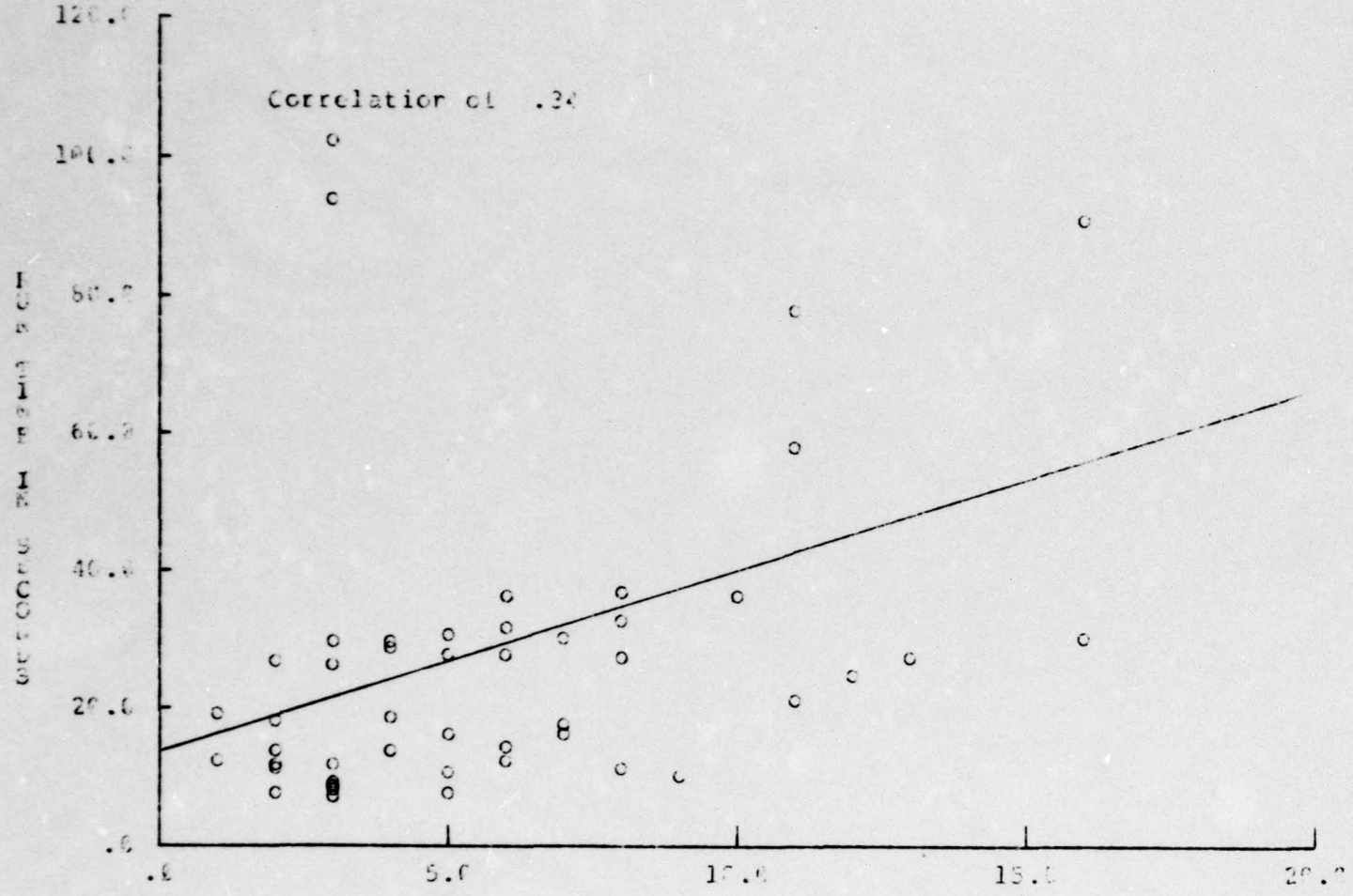
Disk reads, averaging 1,073 input requests per search, were much lower than for the linear search. It is necessary to block the keys and use dump mode I/O to get this value, as the full key file of 12,500 blocks is read for each search question. The median number of disk reads was only 320, the few questions with a high number of false drops running up the average.

The searches averaged 556 drops from the keys of which an average of 480 were false drops. This means that 86 percent of the drops from the search keys were eliminated in the secondary scan of the full record. Another way of looking at this is to measure the false dropping fraction. The false dropping fraction ranged from 0.0 (2 searches had no false drops) to 0.045 (one search had 4,481 false drops) with an average dropping fraction of 0.0048. This average false dropping fraction means that for approximately every 208 keys examined, one will

falsely imply that the record will satisfy the search question. In the median case, out of every 1,639 keys examined one will give a false drop. Since scanning the keys is much faster than looking at the records themselves, not scanning most of the records results in substantial run time savings. As can be seen from Figure 17, the run time shows only a moderate increase as the number of terms in the search is increased. This increase is small because questions with a greater number of terms tend to have more synonym groups, and the added specificity of these groups helps the search. When graphed against the number of synomym groups the run time of the superimposed search decreases slightly with the number of groups (see Figure 18).

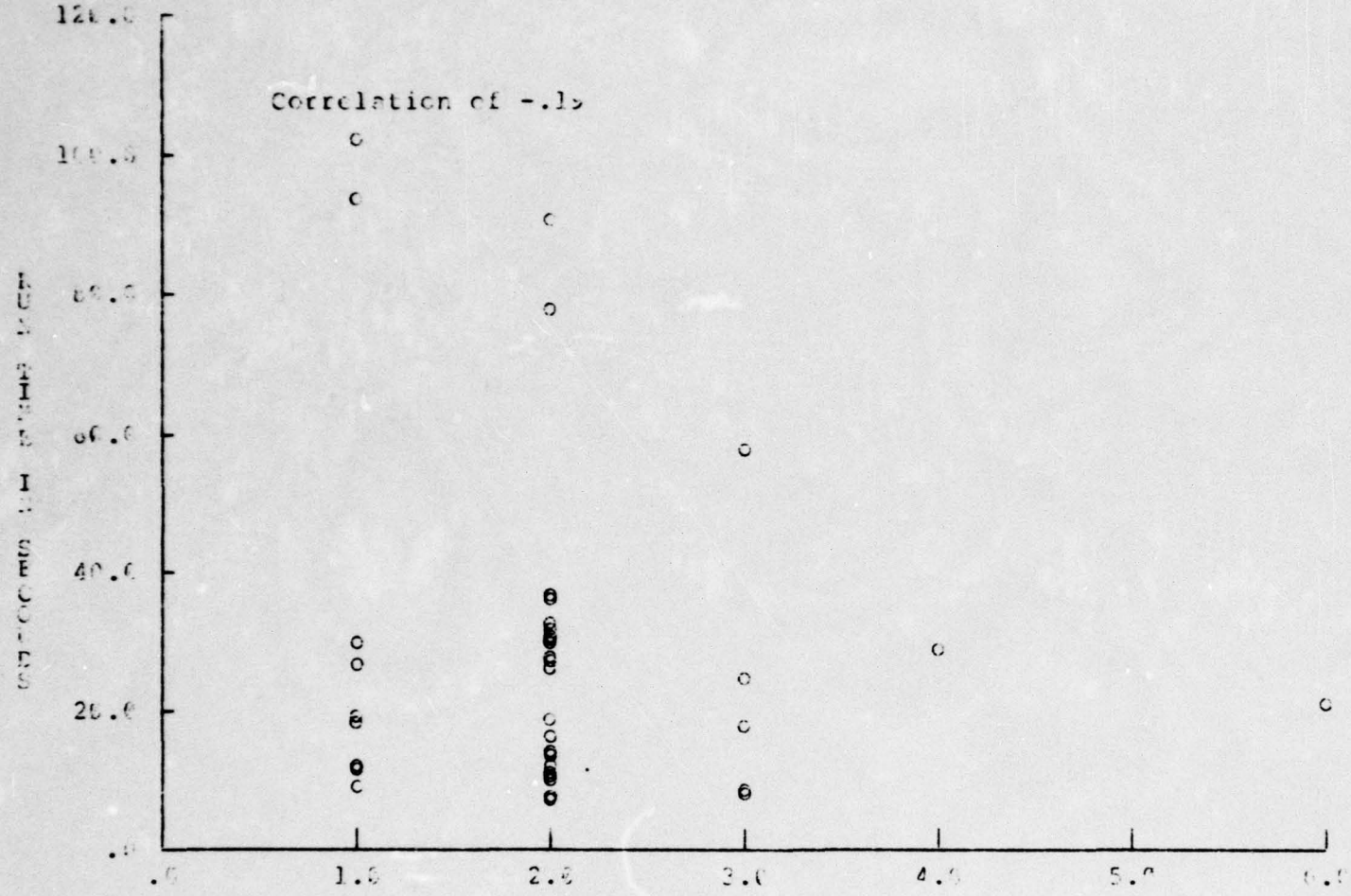
In addition to the file processing required for all search methods, a file of superimposed keys had to be built for the superimposed search. This took 3,132 seconds (52 minutes) run time, or 0.03 seconds per record, about 16 percent of the time needed to initally process the records, so that the increase in time that would be incurred if the keys were generated at the time of the initial processing is small, but not insignificant. The total amount of file processing, including translation for the superimposed search was 22,460 seconds (0.225 seconds per record). It amounts to about four linear searches of the file, i.e. if more than four searches of the file are done it would be less expensive in CPU time to use superimposed coding than to use a linear file.

All of these times are linearly related to the number of records



Superimposed Run Time vs. Number of Search Terms

Figure 17



Dependence of Superimposed Search on Query Complexity

Figure 1>

processed or searched. File setup times and search times for twice as many records would be twice as long for both the linear and superimposed search.

Inverted Search Results

The full set of 339 search questions was run against the inverted file built from the 100,000 INSPEC records. As expected, the inverted search times were much faster than the superimposed search. The average run time was only 1.60 seconds with a standard deviation of 1.65, and a 90 percent confidence interval of 1.47 to 1.76 seconds. The searches had a range of times from 0.07 to 10.3 seconds CPU time, with a median of 1.0 seconds.

Clock time averaged 7.0 seconds per search with a median of 5.1 seconds and a range of 0.2 to 50 seconds. Disk reads averaged 180 per search, with a median of 110 and a range of six to 1,720 disk reads.

It is not as sensible to talk about search time per record for this file structure, because the search times are less related to the number of records than in the other searches. Doubling the file used would only result in a moderate increase in execution time.

File constuction times are, however, closely related to the number of records processed for the inverted file. The inverted file took by far the longest of the three searches to build the files it needs for searching:

Translation:	19,328 seconds
Term separation:	4,670 seconds
Sort of terms:	9,630 seconds
Build inverted file:	4,801 seconds
Total:	38,429 seconds

(0.385 seconds per record)

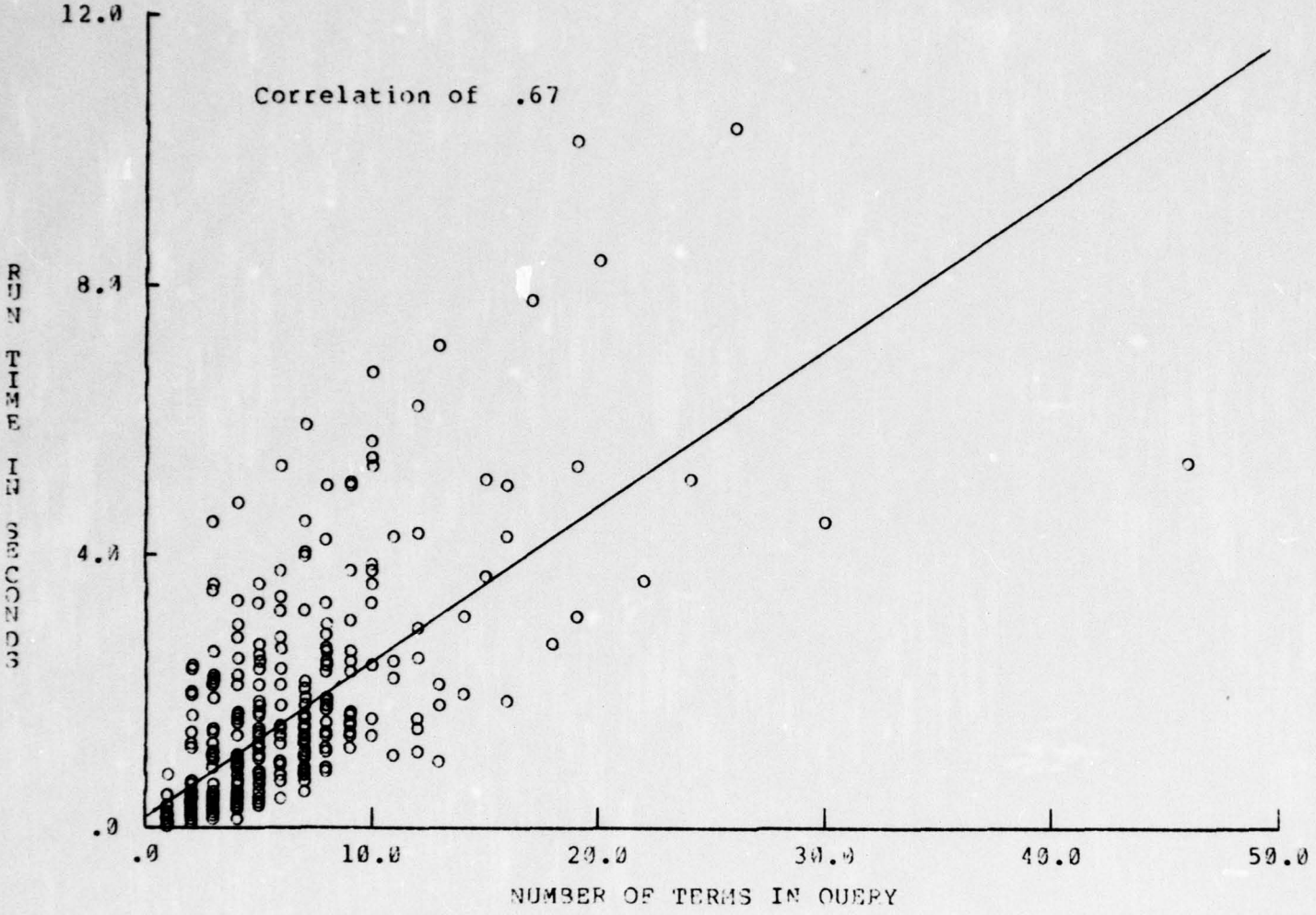
More than 24,000 searches would have to be done on this inverted file before search CPU time exceeded the CPU time taken to prepare the file[6].

As with the superimposed search, the run time was highly correlated with clock time (0.94), and as the analysis showed, very dependant on the number of terms in the search question. See Figure 19 for a graph of the inverted search's run time dependance on number of search terms.

File construction time was 19,101 seconds greater than for a sequential file and 15,969 seconds longer than needed for the superimposed search. These equal 538 superimposed searches and 24 linear searches. In other words, if fewer than 24 searches are to be done on the file it would be cheaper to use a linear file than an inverted one, and if fewer than 538 searches are to be done it would be cheaper to use the superimposed search than the inverted.

---

[6]Chapter VI has a summary table of the run times and other figures for all three searches.



Inverted Search Run Time vs. Number of Search Terms

Figure 19

Search Output Timings

In a test of the time required to format and output records for printing, it took 0.027 CPU seconds per record printed. A more elaborate formating program such as is available on many of the commercial services might well take longer, but should be well under a tenth of a second per record. This is about the time needed to reformat a record for input, ignoring the character code translation and statistics gathering. With an average of 71 hits per search this time would be between two and seven seconds CPU time per search. Since these times would be the same for all of the file structures tested, this does not change their relationship in terms of which is the most economical.

Clock time for output averaged 0.22 seconds per record, or 16 seconds for 71 postings, but this could, and normally would, be done offline and would not affect response time for the searches.

CHAPTER VI

SUMMARY AND CONCLUSIONS

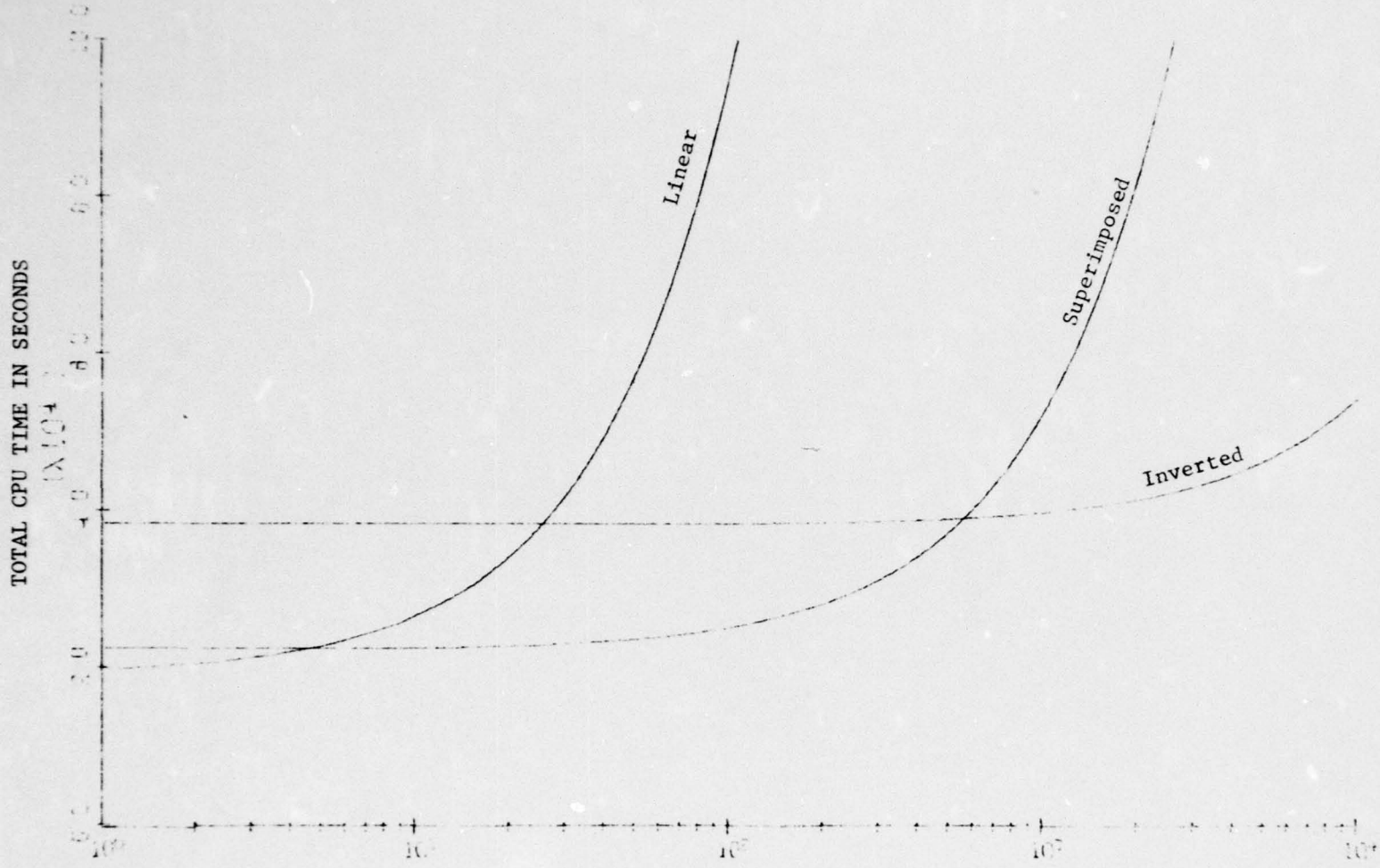
General Findings

It was found that the inverted file takes longer to construct than a superimposed file, and that the faster search possible with the inverted file does not pay for itself until a large number of searches are made of the data base. Figure 20 is a graph showing the relationship of the three searches in terms of CPU time needed to set up the file and perform a given number of searches. If only a small number of searches are done (less than 4) the linear search uses the least amount of CPU time, if a large number are done (more than 500) the inverted search uses the least, but there is an intermediate region in which the superimposed search is the most efficient.

Figure 21 is a summary of the expermental results of the three search algorithms. These results are the basis of Figure 20.

Total Cost of the Searches

These results can be extended from run time to the total cost of providing access to the data base by the three types of searches (see Figure 22 for a graph of this). This total cost is in dollars and is based on rates charged at the University of Illinois. Computer rates vary enormously in both what is charged and how much is charged, but



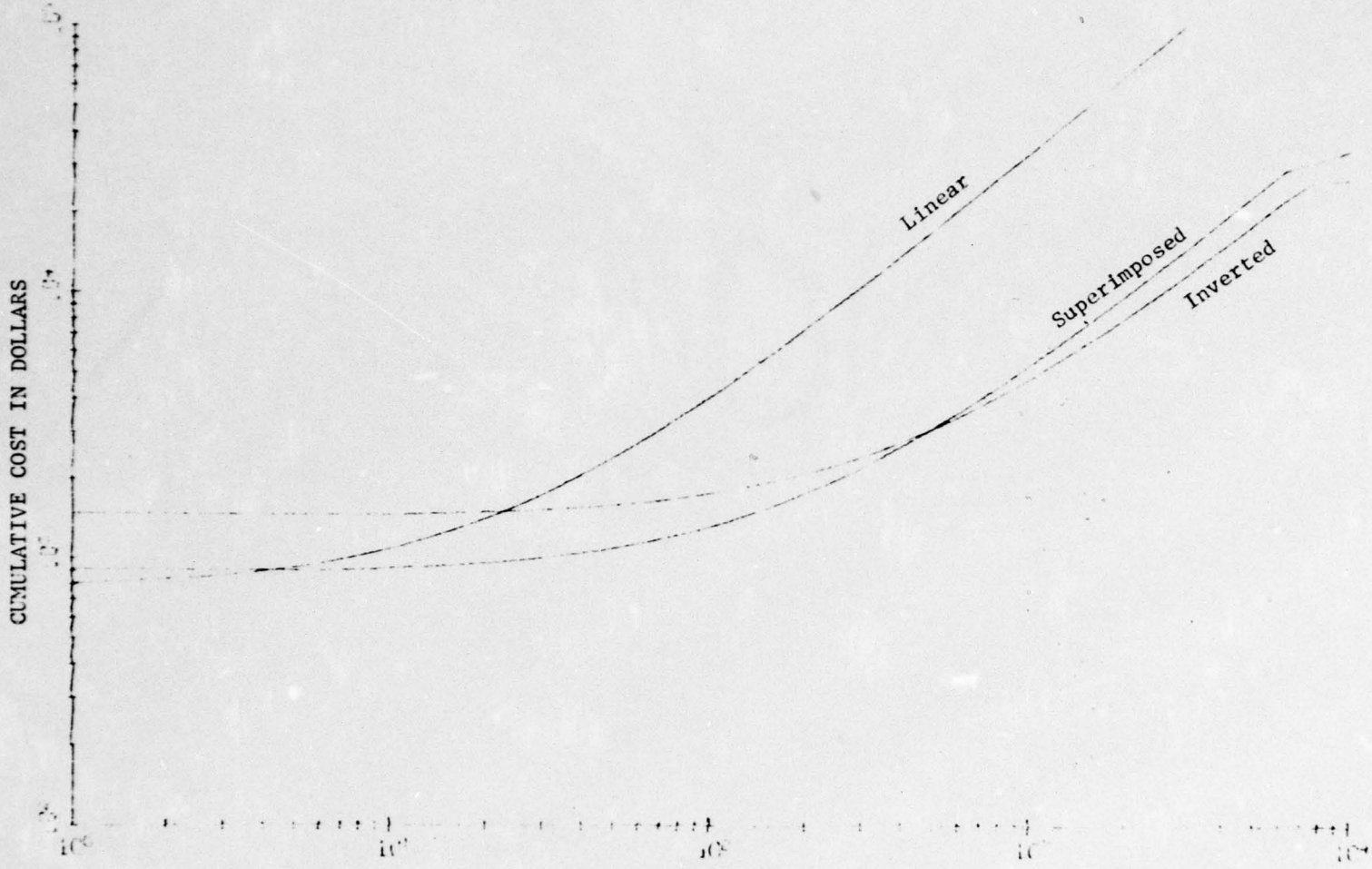
Relative Run Times of the Search Algorithms

Figure 20

	Linear	Superimposed	Inverted
Tape translation & reformatting	19,328 sec.	19,328 sec.	19328 sec.
Term separation	- -	- -	4,470 sec.
Term sort	- -	- -	9,630 sec.
Index generation	- -	3,132 sec.	4,801 sec.
Increase of set up time over Linear	0 %	16 %	99 %
Avg. search CPU time	775 sec.	29.7 sec.	1.6 sec.
Avg. search clock time	957 sec.	95 sec.	7.0 sec.
Avg. search response time	46 sec.	19 sec.	7.0
Avg. disk reads per search in blocks	50,516	1,073	100
Avg. disk blocks per search transferred	50,516	13,475	100
Core used during search	47K	80K	56K
No. of searches run	9	49	339
Avg. no. of false drops	- -	480	- -
Index storage in blocks	0	12,500	44,400
Index storage/ indexed fields (%)	0 %	24 %	87 %
Index storage/ whole INSPEC rec. (%)	0 %	7 %	25 %

Summary of Results

Figure 21



Cost Comparison of Searches

Figure 22

this does give some idea of the relative costs of the searches. CPU and core memory charges are prime time rates as charged on CSL's DECsystem-10, as were printing, online storage costs, and terminal connect time charges. Offline storage and the charge for mounting of disks are based on those charged by the University of Illinois's main computer center. Offline storage costs (on disk) were used until searches become so frequent that the offline storage, plus the mount charge, plus the online storage during the search, becomes greater than continuous online storage. If the total number of searches is spread over a ten month period (approximately the period of time covered by the data base), with searches being performed ten hours per day, this crossover to permanent online storage occurs when the searches occur an average of every half hour.

An estimate of $0.30 was used for the cost of printing the searches. This raises the cost of an individual inverted search from $0.06 to $0.36, and a superimposed search from $1.20 to $1.50. A single liinear search would cost $30.16.

The main cost which is ignored in this estimation is the cost of the data base itself. Since this study is deemed extendable to other data bases with similar characteristics, and since such costs are very data base dependant, this has not been considered. If this had been charged to the searches, it would simply raise all the curves an equal amount on the graph, and would make the costs of the actual searches smaller in comparison to the cost of obtaining and preparing the file.

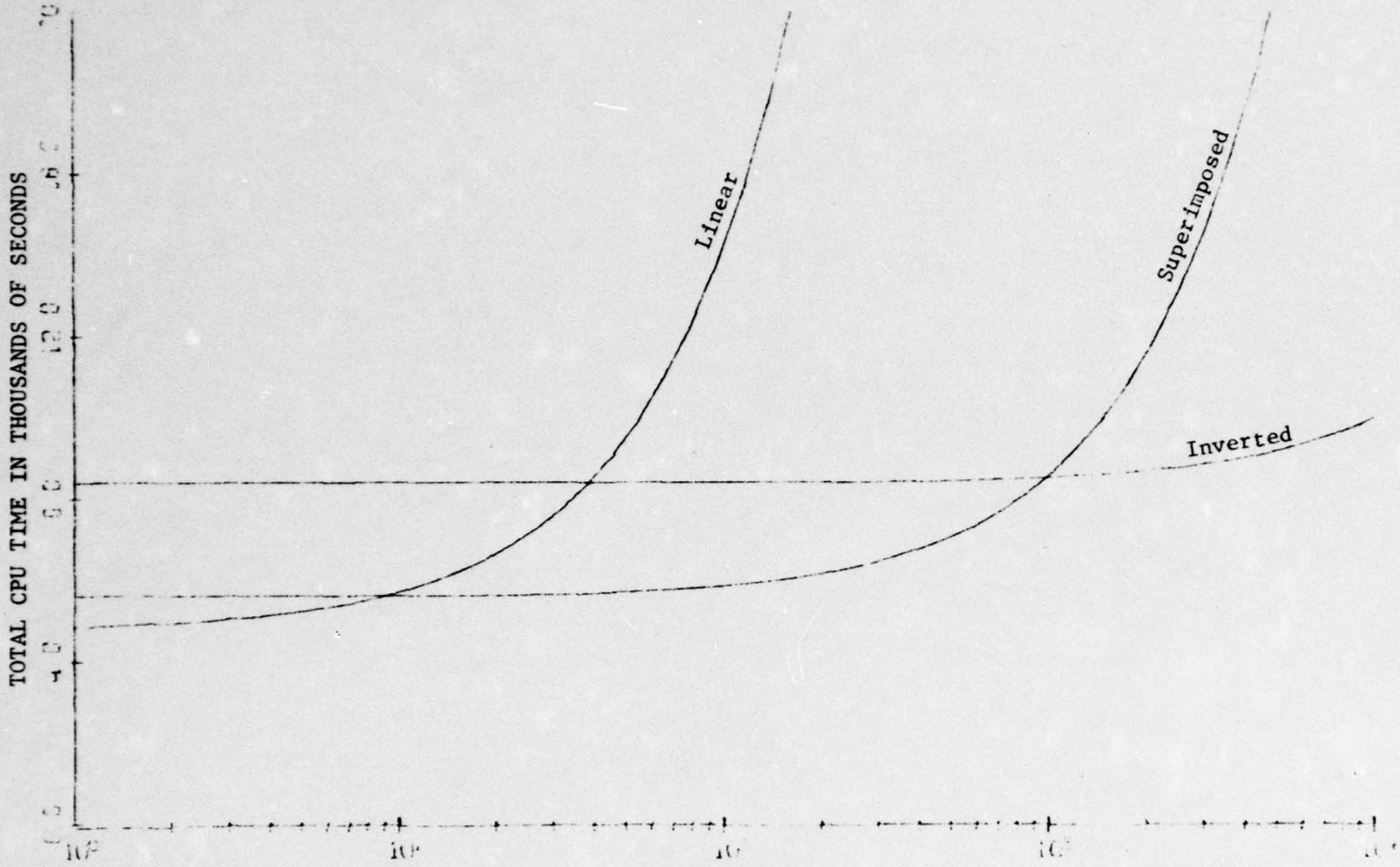
Extension to a Different Computer

The University of Illinois has recently acquired a CYBER 175 manufactured by Control Data Corporation. This is one of the newer computers available, and has an architecture very different from the DECsystem-10 on which this study has been done. A standard benchmark program written in ALGOL has recently been published by Curnow [20], and this was run on both the DECsystem-10 and the CYBER 175. Using the no optimization option on the CDC ALGOL compiler (which produces code of the same level of sophistication as Digital's ALGOL) the CYBER runs slightly more than four times as fast as the DECsystem-10. Other timings on the I/O capabilities of the CYBER show a more startling improvement. Direct access input ran on the CYBER in one-half the clock time, and used only 1/20 the CPU time taken by the DECsystem-10. This is because the CYBER has PPU's (Peripheral Processing Units) which handle all of the I/O. This increase in I/O speed would be most important in the searches, but would also help the term sort associated with building the inverted file. In some systems I/O requests may involve an extra charge which could be significant in file construction and search costs, such as on IBM's 360 and 370 computers where I/O is not charged to the user through CPU time, but as an added charge.

The CYBER lacks instructions specifically designed to handle strings (the DECsystem-10 has several), which would slow down many of these programs, but its longer word size (60-bit instead of the DECsystem-10's 36-bit) would be a definite advantage, especially in the search programs.

It is estimated that the programs used in this study would all run about four times as fast on the CYBER, in terms of CPU time, except for the sorts and linear search, which might run eight times as fast, and the superimposed and inverted searches which would run ten times as fast. These programs would run proportionately faster because they involve a minimum of character processing, and a maximum of disk input, which runs very fast.

For a graph of the estimated relative run times for various numbers of searches of the 100,000 record data base see Figure 23. These are estimates based on limited experience on the CYBER. However there seems to be no way in which any change in these estimates would affect the basic hypothesis that for a certain number of searches the superimposed search is cheaper in total run time than either the linear or inverted file. The increase in file building time of the inverted file over the superimposed keys is slightly less because the sorts which are an important part of the inverted file building are expected to run proportionately faster, however the decrease in run time of all the searches favors the superimposed and linear searches. The linear search becomes feasible for a greater number of searches, and so does the superimposed search. As noted by Lefkovitz [45], faster computers favor searches which now take great amounts of CPU time, and they make it less worthwhile to invert a file for a few searches.



Estimated Run Times on the CYBER 175

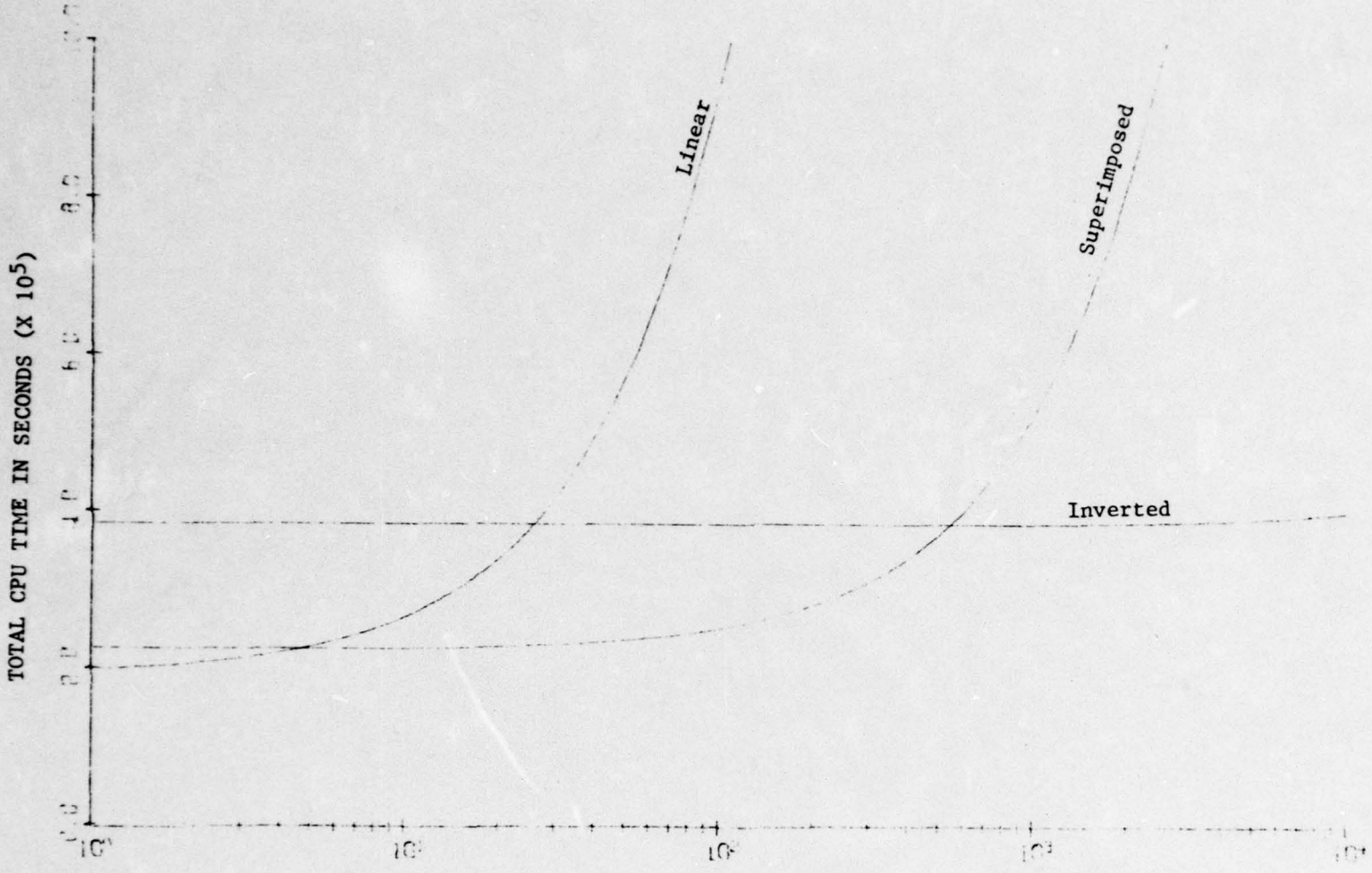
Figure 23

Extensions to Other Data Bases

As the data base becomes larger, search times for the linear and superimposed searches increase linearly with the size of the file. The inverted search has to be on a much larger data base before the linear components of its run time dominate the basic overhead which is present in searches of files of all sizes. Figure 24 shows the projected run times for the three searches on a file 10 times as large as the one used in this study. The inverted search becomes competitive with the superimposed search at a slightly lower number of searches than when the file is smaller.

During the preliminary investigations a number of searches were run on a file of 60,000 titles from ISI's Science Citation Index. These searches are not as good a sample as the searches used in the timings on the INSPEC data base, but do give an indication of the performance of superimposed coding on a simpler, shorter file.

Run times on the ISI data base ranged from 1.0 seconds to 35 seconds CPU time. When normalized to a file of 100,000 records this is about one-fifth the time taken by the INSPEC search by the superimposed search. The shortness of the titles compared with the INSPEC index strings seems to be the most important factor in the shorter run times. On this type of file superimposed coding is more competitive with the inverted search than it is on a file such as INSPEC.



Run Times on a One Million Record Data Base

Figure 24

Limitations and Assumptions

One of the major limitations in this study is the set of search questions used. These questions are only representative of the type of questions asked of one data base at one search center, and searches at other search centers, and especially on other data bases may be very different. A survey done by Martha Williams and Alan Stewart for the Association of Scientific Information Dissemination Centers [80] shows that in number of terms per question the NERAC questions fall within the range found at several other centers. This survey also shows that some centers report as many as 40 to 50 terms in their average search question.

Salton [59] in particular is critical of the pure Boolean type of search algorithm and search question which is a basic factor in the design and testing of these search algorithms. Boolean queries have been the normal mode of search in automated retrieval systems since their inception, however, and will remain so in the foreseeable future.

The data base is less of a limitation, except that the results are restricted to bibliographic data bases. Preliminary testing on LC's MARC and Science Citation Index files showed no problems in applying the superimposed search to them. Different techniques would have to be developed to apply superimposed searching to longer records such as abstracts or full text. It was assumed that the data base is static. Relaxation of this restriction may well reinforce the conclusions of this study, but this has not been demonstrated.

The run times on the DECsystem-10 are representative of the amount of computation needed to setup and search these files on typical computers available today. In the future, radical changes such as extensive parallelism and distributed networks could change this, and these results can not be directly applied to computers with completely different architecture.

It is assumed that computer time is an important variable to be measured in considering what type of search is appropriate for a file. In particular, such matters as programming effort required and type of search made possible by an algorithm can be major considerations in the choice of file structures. An example of this is that the short response time possible with an inverted search may well be a deciding factor in its selection, even though the computer costs for providing this search are higher than the costs of other types of searches. Such matters are highly subjective and application dependant, and were not considered.

Conclusions

It can be concluded that files of bibliographic records containing upwards of a million records can be searched more efficiently by a superimposed search than by either a linear or inverted search if the total number of searches is between 10 and 100 to 500. The hypothesis is therefore accepted.

Practical Implications

Superimposed searching should be useful to people searching moderate sized files (less than one million records) of bibliographic information infrequently (less than once per hour). It may also be useful in dynamic files for which the extensive file maintenance procedures needed are not available to maintain an inverted file, or the cost of regenerating the indexes during updates can not be justified by the amount of searching which is done on the file. The ease in which the superimposed keys can be rearranged may well make them useful in the dynamic library concept of Salton [59]. Many organizations still do linear searches in files of substantial size, although it is difficult to find more than oblique references to such activity in the literature. A superimposed search is a much more efficient file design for searching than a simple linear file, and is an alternative which can be implemented fairly easily, with the file maintained in much the same way in which a linear file is.

It is especially true of the minicomputer and microcomputer systems now becoming widely available that an inverted file system is not available from the vendor of the computer, and sufficient mass storage is not available for an efficient implementation of an inverted file. The slowness of microcomputer systems can make a linear search of a file of any substantial size take hours to accomplish. This may be an environment in which superimposed searches will be useful, and may even become widespread.

Suggestions for Further Research

Application of superimposed searching to full text searching is an area that should be investigated. As computers become faster, as storage costs continue to decrease, and as more computer editing replaces paper editing, the full texts of much more material will become available in machine readable form and will therefore be searched by computer. Superimposed keys may offer an attractive compromise between full inversion of the file and no indexes at all.

New computer hardware becoming available in the next few years will have a profound effect on all computer operations. File searching is a type of computing in which parallelism in computers could be very useful, and networks of relatively inexpensive minicomputers will have an effect on the types of file searching that is practical. Dedicated minicomputers and microcomputers make run time less important than clock time since only one job is being run at a time. The effect of this and the restricted mass storage available on the types of searches that are most useful on systems of this type should be investigated.

It would be very useful to analyze of the advantages of using equifrequent substrings instead of n-grams to build the keys. Would there be an appreciable increase in the entropy of the keys? Equally important would be an analysis of how much extra computation would be needed to use these substrings to generate the keys.

It is possible that more complex keys with length dependant on the length of string they index could reduce both the length of the key file

and the false dropping fraction. Investigation of the effect of such keys on overall run time should be considered.

BIBLIOGRAPHY

[1] Adamson, George W., and Boreham, William. "The Use of an Association Measure Based on Character Structure to Identify Semantically Related Pairs of Words and Document Titles." Information Storage and Retrieval 10 (July/August 1974): 253-260.

[2] Aho, Alfred V.; Hopcroft, John E.; and Ullman, Jeffrey D. The Design and Analysis of Computer Algorithms. Reading MA: Addison-Wesley, 1974.

[3] Aho, Alfred V. and Corasick, Margaret J. "Efficient String Matching: An Aid to Bibliographic Search." Communications of the ACM 18 (June 1975): 333-340.

[4] Aitchison, T. M., and Tracy, Jennifer M. "Comparative Evaluation of Index Languages--Part I: Design." July 1969, Institution of Electrical Engineers, London, England. (Report No. INSPEC/4).

[5] Aitchison, T. M.; Hall, Angela M.; Lavelle, Katherine H.; and Tracy, Jennifer M. "Comparative Evaluation of Index Languages--Part II: Results." July 1970, Institution of Electrical Engineers, London, England. (Report No. INSPEC/5).

[6] American National Standards Institute. American National Standard for Bibliographic Information Interchange on Magnetic Tape. American National Standards Institute, New York NY, 1971, 34p. (ANSI-Z39.2-1971).

[7] Barton, Ian J; Creasey, Susan E.; Lynch, Michael F.; and Snell, Michael J. "An Information Theoretic Approach to Text Searching in Direct Access Systems." Communications of the ACM 17 (June 1974): 345-350.

[8] Bird, P. R. "Design Analysis of Random Superimposed Coding Methods for Data Storage." Information Processing and Management 11 (August 1975): 79-88.

[9] Bloom, Burton H. "Some Techniques and Trade-Offs Affecting Large Data Base Retrieval Times." In:Association for Computing Machinery 24th National Conference, San Fransisco CA, August 1969. Proceedings of the ACM 1969. Association for Computing Machinery, New York NY 1969, 83-95.

[10] Bookstein, Abraham. "On Harrison's Substring Testing Technique." Communications of the ACM 16 (March 1973): 180-181.

[11] Bookstein, Abraham. "A Hybrid Access Method for Bibliographic Records." Journal of Library Automation 7 (June 1974): 97-104.

[12] Bourne, Charles P., and Ford, Donald F. "A Study of the Statistics of Letters in English Words." Information and Control 4 (March 1961): 48-67.

[13] Burke, J. Michael, and Rickman, J. T. "Bitmaps and Filters for Attribute-Oriented Searches." International Journal of Computer and Information Sciences 2 (September 1973): 187-200.

[14] Burkhardt, Walter H. "An Efficient File-Organization for Frequency-Distributed Retrieval." In: American Society for Information Science 36th Annual Meeting, Los Angeles CA, 21-25 October 1969. Proceedings, vol. 10. Innovative Developments in Information Systems: Their Benefits and Costs. Helen J. Waldron and F. Raymond Long, eds. Greenwood Press, Westport CN. 1973, 29-30.

[15] Byrne, J. G.; Currivan, P. J.; and Mahon, F. V. "A Current Awareness System Based on INSPEC Tapes." Information Storage and Retrieval 8 (August 1972): 177-190.

[16] Cardenas, Alfonso F. "Evaluation and Selection of File Organization--A Model and System." Communications of the ACM 16 (September 1973): 540-548.

[17] Cardenas, Alfonso F. "Analysis and Performance of Inverted Data Base Structures." Communications of the ACM 18 (May 1975): 253-263.

[18] Cardenas, Alfonso F., and Sagamang, James P. "Modeling and Analysis of Data Base Organization; The Double Chained Tree Structure." Information Systems 1 (April 1975): 57-67.

[19] Crouch, C. J. "An Evaluation of Languages for the Implementation of Information Storage and Retrieval Systems." ACM SIGPLAN Notices 10 (January 1975): 113-117.

[20] Curnow, H. J., and Wichmann, B. A. "A Synthetic Benchmark." The Computer Journal 19 (February 1976): 43-49.

[21] De Heer, T. "Experiments with Syntactic Traces in Information Retrieval." Information Storage and Retrieval 10 (March/April 1974): 133-144.

[22] Dodd, George G. "Elements of Data Management Systems." Computing Surveys 1 (June 1969): 117-133.

[23] Doszkocs, Tamas E. "Analysis of Term Distribution in the TOXLINE Inverted File." Journal of Chemical Information and Computer Sciences 16 (August 1976): 131-135.

[24] Elias, Peter. "Efficient Storage and Retrieval by Content and Address of Static Files." Journal of the Association for Computing Machinery 21 (April 1974): 246-260.

[25] Elias, Peter, and Flower, Richard A. "The Complexity of Some Simple Retrieval Problems." Journal of the Association for Computing Machinery 22 (July 1975): 367-379.

[26] Ehrhardt, F. "German Computer Services Based on CA Condensates." In: European Association of Scientific Information Dissemination Centres. Conference. Luxembourg, 18-19 October 1972. Proceedings. Better Service for the User. Directorate General for Dissemination of Information, Centre for Information and Documentation, Luxembourg, June 1973, 21-25.

[27] Feldman, Alfred. "An Efficient Design for Chemical Structure Searching. I. The Screens." Journal of Chemical Information and Computer Science 15 (August 1975): 147-152.

[28] Fokker, Dirk W., and Lynch, Michael F. "Application of the Variety-Generator Approach to Searches of Personal Names in Bibliographic Data Bases--Part 1. Microstructure of Personal Authors' Names." Journal of Library Automation 7 (June 1974): 105-118.

[29] Fokker, Dirk W., and Lynch, Michael F. "Application of the Variety-Generator Approach to Searches of Personal Names in Bibliographic Data Bases--Part 2. Optimization of Key-Sets, and Evaluation of Their Retrieval Efficiency." Journal of Library Automation 7 (September 1974): 201-213.

[30] Foster, Caxton C. Content Addressable Parallel Processors. New York: Van Nostrand Reinhold, 1976.

[31] Goble, C. E. "A Free Text Retrieval System Using Hash Codes." The Computer Journal 18 (February 1975): 18-20.

[32] Harrison, Malcom C. "Implementation of the Substring Technique by Hashing." Communications of the ACM 14 (December 1971): 777-779.

[33] Heaps, H. S., and Thiel, L. H. "Optimum Procedures for Economic Information Retrieval." Information Storage and Retrieval 6 (June 1970) 137-153.

[34] Higgins, L. D., and Smith, F. J. "Disc Access Algorithms." The Computer Journal 14 (August 1971): 249-253.

[35] Hsiao, David. "A Formal System for Information Retrieval from Files." Communications of the ACM 13 (February 1970): 67-73.

[36] Hutton, Fred C. "PEEKABIT, Computer Offspring of Punched Card PEEKABOO, for Natural Language Searching." Communications of the ACM 11 (September 1968): 595-598.

[37] Iker, Howard P. "Solution of Boolean Equations Through Use of Term Weights to the Base Two." American Documentation 18 (January 1967): 47.

[38] Katzer, Jeffrey. "The Cost-Performance of an On-Line, Free-Text Bibliographic Retrieval System." Information Storage and Retrieval 9 (June 1973): 321-329.

[39] King, D. R. "The Binary Vector as the Basis of an Inverted Index File." Journal of Library Automation 7 (December 1974): 307-314.

[40] Knott, G. D. "Hashing Functions." The Computer Journal 18 (August 1975): 265-278.

[41] Knuth, Donald E. The Art of Computer Programming. Vol. 3: Sorting and Searching. Reading, MA: Addison-Wesly, 1973.

[42] Knuth, Donald E.; Morris, James H. Jr.; and Pratt, Vaughan R. Fast Pattern Matching in Strings. Stanford: Stanford University Computer Science Department, 1974. (STAN-CS-74-440).

[43] Lancaster, F. W., and Fayen, E. G. Information Retrieval On-Line. Los Angeles: Melville, 1973.

[44] Lefkovitz, David. File Structures for On-Line Systems. New York: Spartan Books, 1969.

[45] Lefkovitz, David. "The Large Data Base File Structure Dilemma." Journal of Chemical Information and Computer Sciences 15 (February 1975): 14-19.

[46] Lipetz, Ben-Ami. "Influence of File Activity, File Size, and Probability of Successful Retrieval on Efficiency of File Structures." In: American Society for Information Science 32nd Annual Meeting, San Francisco CA, 1-4 October 1969. Proceedings, vol. 6. Cooperating Information Societies. Jeanne B. North, ed. Greenwood Publishing Corp., Westport CN 1969, 175-179.

[47] Lowe, Thomas C. "The Influence of Data Base Characteristics and Usage on Direct Access File Oganization." Journal of the Association for Computing Machinery 15 (October 1968): 535-548.

[48] Lum, V. Y.; Ling, H.; and Senko, M. E. "Analysis of a Complex Data Management Access Method by Simulation Modeling." In: American Federation of Information Processing Societies Conference, Houston TX, 17-19 November 1970. Proceedings, vol. 37. 1970 Fall Joint Computer Conference. AFIPS Press, Montvale NJ 1970, 211-222.

[49] Lynch, Michael F. "Creation of Bibliographic Search Codes for Hash-Addressing using the Variety-Generator Method." Program 9 (April 1975): 46-55.

[50] Lynch, Michael F.; Petrie, Howard; and Snell, Michael J. "Analysis of the Microstructure of Titles in the INSPEC Data-Base." Information Storage and Retrieval 9 (June 1973): 331-337.

[51] McCreight, Edward M. "A Space-Economical Suffix Tree Construction Algorithm." Journal of the Association for Computing Machinery 23 (April 1976): 262-272.

[52] Manacher, Glenn K. "A Benchmark Calculation for Retrieval by Confluence of Binary Attributes in a Random-Access Computer." American Documentation 20 (April 1969): 168-169.

[53] Martin, Thomas H., and Parker, Edwin B. "Comparative Analysis of Interactive Retrieval Systems." ACM SIGPLAN Notices 10 (January 1975): 75-85.

[54] Mooers, Calvin N. "Zatocoding Applied to Mechanical Organization of Knowledge." American Documentation 2 (January 1951): 20-32.

[55] Mooers, Calvin N. "Choice and Coding in Information Retrieval Systems." Transactions of the I. R. E. Profesional Group on Information Theory PGIT-4 (Septmber 1954): 112-118.

[56] Price, C. E. "Table Lookup Techniques." Computing Surveys 3 (June 1971):49-65.

[57] Reardon, Bernard C. "An Adaptive Information Retrieval System Using Partial File Inversion." Information Storage and Retrieval 10 (February 1974): 49-56.

[58] Rettenmayer, John W. "File Ordering and Retrieval Cost." Information Storage and Retrieval 8 (April 1972): 79-93.

[59] Salton, Gerald. Dynamic Information and Library Processing. Englewood Cliffs NJ, Prentice-Hall 1975.

[60] Schipma, Peter B. "Computer Search Center Statistics on Users and Data Bases." Journal of Chemical Documentation 14 (February 1974): 25-29.

[61] Schuegraf, E. J. "Selection of Equifrequent Word Fragments for Information Retrieval." Information Storage and Retrieval 9 (December 1973): 697-711.

[62] Schuegraf, E. J., and Heaps, H. S. "A Comparison of Algorithms for Data Base Compression by Use of Fragments as Language Elements." Information Storage and Retrieval 10 (September/October 1974): 309-319.

[63] Severance, Dennis G. "Identifier Search Mechanisms: A Survey and Generalized Model." Computing Surveys 6 (September 1974): 175-194.

[64] Severance, Dennis G. "A Parametric Model of Alternative File Structures." Information Systems 1 (April 1975): 51-55.

[65] Severance, Dennis G., and Merten, Alan G. "Performance Evaluation of File Organizations Through Modelling." In: Association for Computing Machinery Annual Conference, Boston MA, August 1972. Proceedings. Association for Computing Machinery, New York NY 1972, 1061-1072.

[66] Shannon, C. E. "A Mathematical Theory of Communication." Bell System Technical Journal 27 (July and October 1948): 379-423 and 623-656.

[67] Stiassny, Simon. "Mathematical Analysis of Various Superimposed Coding Methods." American Documentation 11 (April 1960) 155-169.

[68] Swid, R. E. "Linear vs. Inverted File Searching on Serial Access Machines." In: American Documentation Institute 26th Annual Meeting, Chicago Il, 6-11 October 1963. Proceedings part II, H. P. Luhn ed. American Documenation Institute, Washingtion DC 1963 pp. 169-170.

[69] Thiel, L. H., and Heaps, H. S. "Program Design for Retrospective Searches on Large Data Bases." Information Storage and Retrieval 8 (February 1972): 1-20.

[70] Uhlmann, W. "The Application of Random Superimposed Coding and Chain Spelling to Peek-A-Boo Cards." American Documentation 15 (April 1964): 89-92.

[71] Ulmann, Wolfram. "A General Method for Matching Arbitrary Logical Statements in Mechanized Retrieval Systems." American Documentation 20 (July 1969): 253-258.

[72] Vallarino, Oscar. "On the Use of Bit Maps for Multiple Key Retrieval." In: Association for Computing Machinery Special

Interest Group on Programming Languages (SIGPLAN) and Special Interest Group on Management of Data (SIGMOD) Conference on Data Abstraction, Definition, and Structure. Salt Lake City UT, 22-24 March 1976. 8 SIGPLAN Notices (Special issue II 1976). Association for Computing Machinery, New York NY 1976, 108-114.

[73] Vose, M. R., and Richardson, J. S. "An Approach to Inverted Index Maintenance." The Computer Bulletin 16 (May 1972): 256-262.

[74] Warheit, I. A. "File Organization of Library Records." Journal of Library Automation 2 (March 1969): 21-30.

[75] Weiner, Peter. "Linear Pattern Matching Algorithms." In: The Institute of Electrical and Electronics Engineers Computer Society and the Univeristy of Iowa, Annual Symposium on Switching and Automata Theory. Ames IO, October 1973. Proceedings vol. 14. Institute of Electrical and Electronics Engineers Computing Society, Northridge CA 1973, 1-11. (IEEE Publication No. 73CHO 786-4-C).

[76] Wilde, Daniel U. "Computerized Chemical Information Retrieval Techniques." Journal of Chemical Documentation 15 (August 1975) 183-185.

[77] Wilde, Daniel U. and Starke, Albert C. "A Chemical Search System for a Small Computer." Journal of Chemical Documentation 14 (February 1974) 41-44.

[78] Williams, Martha E. "Experiences of IIT Research Institute in Operating a Computerized Retrieval System for Searching a Variety of Data Bases." Information Storage and Retrieval 8 (April 1972): 57-75.

[79] Williams, Martha E.; Schipma, Peter B.; Preece, Scott E.; Becker, David S.; Lewellwen, Patricia A.; and Stewart, Alan K. Four Year Summary, Educational and Commercial Utilization of a Chemical Information Center. Chicago: IIT Research Institute Computer Search Center, July 1972. (ED 068 132).

[80] Williams, Martha E. and Stewart, Alan K. ASIDIC Survey of Information Center Services, IIT Research Institute, Chicago IL, June 1972, 117p.

[81] Zipf, George Kingsley. Human Behavior and the Principal of Least Effort, An Introduction to the Human Ecology. Reading MA, Addison-Wesley, 1949.

APPENDIX--A

INSPEC FIELD DEFINITIONS

MAIN CATEGORY 0 (CONTROL FIELDS)
001 Control Number
010 Record Type

MAIN CATEGORY 1 (SUBJECT DELINEATION)
100 Title of Record
110 Text of Abstract
120 Sectional Classification Codes
121 Unified Classification Codes
130 Subject Index Headings
131 Free-Indexing Terms
132 Treatment Codes
150 Title of Corresponding Higher Level Publication From Which this Item has Been Taken
151 Title of Cover-to-Cover Translation Journal
160 Language
170 Title of Conference

MAIN CATEGORY 2 (PERSONAL NAMES)
200 Author(s)
210 Editor(s)
220 Translator(s)

MAIN CATEGORY 3 (IDENTIFYING CODES)
300 Abstract Number(s)
310 CODEN
311 CODEN of Cover-to-Cover Translation
320 Standard Book Number
330 Report Number
340 U.S. Government Clearing House Number
350 Contract Number
360 Patent Number
370 Original Patent Application Number

MAIN CATEGORY 4 (VOLUME AND ISSUE)
400 Volume and Issue Number
401 Volume and Issue Number of Cover-to-Cover Translation
450 Part number

MAIN CATEGORY 5 (LOCATIONS)
500 Location of Conference
510 Place of Publication
520 Country of Patent
530 Country of Original Patent Application

MAIN CATEGORY 6 (PAGINATION)
600 Number of Pages of Level 1 Record
610 Number of Pages of Level 2 Record
620 Inclusive Page Numbers
621 Inclusive Page Numbers of Level 2 Cover-to-Cover Translation
630 Number of References
640 Description of Unconventional Medium

MAIN CATEGORY 7 (ORGANIZATIONS)
700 Author Affiliation
710 Editor Affiliation
730 Assignees
740 Publisher
750 Organization Issuing Report
760 Sponsoring Organization
770 Availability

MAIN CATEGORY 8 (DATES)
800 Inclusive Dates of Conference
810 Date of Publication
811 Date of Publication of Cover-to-Cover Translation
820 Date Filed or Submitted
830 Priority Date

MAIN CATEGORY 9 (FILE DESCRIPTION)
900 Identification
910 Destination
920 Date Written
930 Selection Criteria

Note: The items in category 9 appear only in the file header.

APPENDIX--B

SEARCH USER'S MANUAL

Searching INSPEC with
INVSER
LINSER
&
MICSER

A Manual by
Thomas Hickey
CSL - - IRRL
University of Illinois
July 1976

Introduction

There are three programs for searching the INSPEC data base: INVSER, LINSER, and MICSER. INVSER is the program for searching the inverted version of the file, LINSER searches the file character by character, and MICSER uses a superimposed key to do a fast sequential search of the data base.

The INSPEC data base available for searching consists of 100,000 records (covering a period of about nine months) from the 1974 Science Abstracts. The programs used for searching the data base are designed so that the interface with the user is as uniform as possible, although some commands (for example the index display command in INVSER) are available only in one or two of the programs.

At nearly any point in the programs the character X will exit the user from what he is are doing at the moment, and at any point where the program is waiting for input from the terminal, typing the character H should result in a short helpful message of explanation. All commands and searches are recorded in a log file, since these programs are experimental and their performance and operations need to be closely monitored.

Pages 122 through 124 give instructions on how to run individual programs, and pages 125 through 130 are detailed descriptions of the commands available and their use.

Inverted Search

INVSER, the inverted search program, gives the best response of the the three programs and its operation is quite similar to some of the nationally known search programs. After logging in on CSL's PDP-10 the first step is to ensure that the disk which contains INVSER and its files is available. To do this type Mount IRP3:<carriage return> The operator should then mount the disk, and you will get a message when this is acomplished. Execute the program by typing Run INVSER[714,163] <carriage return> The program will respond with "Please identify yourself: " at which point please type your name or other identification of the search for the log file. The program responds to this with "C,D,H,I,R,S or ^X:" meaning that it expects one of these characters to be typed in as a command. A carriage return is not expected at this point. By typing H you will find that the letters stand for Combine searches, Display a search, Help, Index display, Recap commands, Search, and Exit from the program. Other sections of this documentation give detailed explanations of these commands.

Linear Search

LINSER is identical in operation to MICSER. To run it, first mount IRP5 by typing Mount IRP5: <carriage return>. After the disk is mounted type Run LINSER[714,163]. Since the program takes nearly 15 minutes to complete a search the same commands are available that are described in MICSER to check on the search's progress at any point.

Superimposed Search

To run the superimposed search, first type Mount IRP5:<carriage return> then Run MICSER[714,163] <carriage return> and go through the identification process as explained in the inverted search section. Since MICSER does not use an index like INVSER's, the index display command (I) is not available.

Although faster than LINSER, MICSER's response time is normally on the order of one to three minutes for a complete search. To combat this, three additional commands are available during the search phase. After the search has started, typing the character S will "Show" the status of the search--hits, false drops, and number of references examined so far. Typing an H will result in the type out of hits as they are found, and a F will start the type out of false drops as they occur. A second H or F will stop the type out. A X during the search will exit from it, saving the search results up to that point.

Combine Search Command

The C command allows you to combine previous searches in any Boolean combination you desire, and is especially useful with INVSER, since several short searches on different terms can be done in a short time. Use + to OR searches, * to AND them and \ to NOT them. For example if you have done search number 1 on Libraries, search number 2 on Librarians, and search number 3 on computers 4=(1+2)*3 will result in search number 4 holding all references indexed by libraries and computers, and all those indexed by librarians and computers. Typing simply (1+2)*3 would give the same result if search number 4 has not previously been used, since a number is automatically assigned if none is given.

At present there is an upper limit of 40 on the number of searches that may be held at one time. If this is exceeded an error message will be given, and the program must be restarted.

Display Search Command

The Display command (D) displays the results of searches either on the screen, or to the printer. The simplest form of the command is to type D and after the prompt "->" to type the number of the search that you wish displayed on the screen. To send a search to the printer type #:P:Name where # is the number of the search, and Name is whatever you would like to name the search for print out (up to six letters are used). Both during typeout and print out typing an X will exit from the command.

As a debugging feature, typing #:A will type out only the posting numbers of the search, where # again stands for an actual search number. The command for full typeout of the search is #:B and typing just the number of the search defaults to this mode. Due to space limitations a shortened reference is all that is available for display during INVSER, while during MICSER and LINSER only the index string can be displayed.

Exit Command

When the program is waiting for a command a ^X (Control X) will exit from the program. A ^X is used at this point instead of a simple X to prevent inadvertent exiting from the search system. ^C's have the same effect as ^X, but are more general since they may be done at any point in the program, not just when the program is expecting input from the terminal.

Recap Command

After several searches have been made it is often very useful to be able to display each search command. The Recap (R) command will display either a single previous command, or a range of commands. Type R for Recap and when the program prompts with a "->" type either a single number (e.g. 13) or a range of numbers (e.g. 3:15).

Search Command

S is the most important command of all. Search questions are input in "product of sums" form e.g.

[LIBRAR#+INFORMATION CENTER]*[AUTOMATION+COMPUTERS]

Upper and lower case letters are equivalent during the search. The # sign may be used on either end of a search term to denote left or right truncation (only right truncation is allowed in INVSER). The + sign stands for OR, the * sign for AND and the \ sign for NOT. Phrases, such as INFORMATION CENTER in the above example are allowed. In addition, any search phrase may be preceded by a tag. An author search for anyone named Smith might be A:Smith,#. The allowed tags are:

C: Controlled Index Phrase

F: Free Index Phrase

U: Uniform Classification

J: Coden

S: Sectional Classification

E: Editor

Each Search is automatically given a number which allows you to refer to it later, to display the references, or to combine it with other searches.

Index Display Command

The index display command is available only to INVSER, since this is the only program which uses a displayable index. After typing I to get the prompt "->", type the term you would like to see displayed. Search tags, if present, are ignored. The program will search the index and display a section around the terms with the terms' frequencies of occurrence. It should be noted that the frequency of occurrence displayed is not necessarily the number of postings a term has, since multiple occurrences of a term within a single reference will each be totaled in this count, but of course are merged into one posting when read in.

VITA

The author, Thomas Hickey, was born on July 30, 1947 in Buffalo New York. He attended elementary and high school in Honeoye Falls, New York, graduating in 1965. After receiving a B.S. in Physics from SUNY at Stony Brook in 1969 he entered the masters program in Library and Information Science at SUNY College at Geneseo, from which he received a M.L.S. in 1970. From 1970 he was employed by the John Crerar Library as a Science Reference Librarian until 1973 when he began the doctoral program in the Graduate School of Library Science at the University of Illinois.

While completing his Ph.D. he has worked as a Graduate Teaching Assistant for the University of Illinois Graduate School of Library Science and as a Graduate Research Assistant for the Information Retrieval Research Laboratory at the Coordinated Science Laboratory of the University of Illinois. He is a member of the Association for Computing Machinery and the American Society for Information Science.